AFIT/GE/ENG/92D-23

FACE RECOGNITION WITH NEURAL NETWORKS

THESIS

Dennis Lee Krepp
Captain, USAF

AFIT/GE/ENG/92D-23

Approved for public release; distribution unlimited

AFIT/GE/ENG/92D-23

FACE RECOGNITION WITH NEURAL NETWORKS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Dennis Lee Krepp, B.S.E.E.

Captain, USAF

December, 1992

Approved for public release; distribution unlimited

Acknowledgments

I would like to thank my thesis advisor, Dr. Steve Rogers, and the rest of my thesis committee, Dr. Dennis Ruck, Dr. Matthew Kabrisky, and Dr. Mark Oxley. Their help made this thesis possible and the experience rewarding. A special thank you goes to Dr. Ruck who provided the backpropagation software that was the foundation for all the programming that followed.

I would like to thank Capt (now Dr.) Kevin Priddy and Capt Ken Fielding for their answers to my many questions. I want to thank the rest of the *Face Group*, Capt Kevin Gay and Capt Ken Runyon, for their support and humor during this research effort. Their friendship will be treasured always. I also wish to to thank Dan Zambon and Dave Doak for keeping the computers up and running at all hours.

Finally, I want to thank my family, especially my lovely wife, Karen, for their support during the past year and a half. Without them, none of this would have been possible.

Dennis Lee Krepp

Table of Contents

Page

Acknowledgments ii

Table of Contents iii

List of Figures vii

List of Tables viii

Abstract ix

I. **Problem Description** 1-1

1.1 **Introduction** 1-1

1.2 **Background** 1-2

1.2.1 **Turk and Pentland at the MIT Media Lab.** 1-2

1.2.2 **Face Recognition at AFIT.** 1-2

1.2.3 **Cottrell at UCSD.** 1-3

1.2.4 **Los Alamos National Laboratory.** 1-3

1.3 **Problem Statement** 1-3

1.4 **Research Objectives** 1-4

1.5 **Assumptions** 1-4

1.6 **Scope and Limitations** 1-5

1.7 **Standards** 1-5

1.8 **Approach/Methodology** 1-5

1.9 **Overview** 1-5

Page

II. Literature Review 2-1

2.1 Introduction 2-1

2.2 The Biology of Face Recognition 2-2

2.3 Feature Extraction and Data Compression 2-4

2.3.1 Principal Component Analysis (PCA). 2-4

2.3.2 PCA using a Linear Neural Network. 2-5

2.3.3 Feature Extraction and Data Compression in Nonlinear Networks 2-9

2.3.4 Karhunen-Loève Network for Feature Reduction 2-12

2.4 Neural Networks for Face Recognition 2-14

2.4.1 Cottrell Neural Network. 2-14

2.4.2 Backpropagation Neural Network for Face Verification. . . 2-16

2.4.3 Additional Network Research 2-19

2.5 Summary 2-19

III. Methodology 3-1

3.1 General 3-1

3.2 Algorithm Development 3-2

3.3 Feature Extraction and Classification Using Identity Networks . . . 3-3

3.4 Verification and Classification Using Backpropagation 3-5

3.4.1 Single Day 3-5

3.4.2 Multiple Day Verification 3-6

3.4.3 Multiple Person Recognition 3-6

3.5 Neural Network Classification versus Karhunen-Loève Transform . 3-7

3.6 Code Development 3-8

3.7 Summary 3-8

Page

IV. Results 4-1

4.1 General 4-1

4.2 Feature Extraction and Classification Using Identity Networks . . . 4-1

4.2.1 Feature Extraction 4-1

4.2.2 Classification Using Identity Network Extracted Features . 4-3

4.2.3 Identity Networks and Multiple Day Data Classification . . 4-7

4.3 Verification and Classification Using Backpropagation 4-8

4.3.1 Single Day Verification 4-8

4.3.2 Multiple Day Verification 4-10

4.3.3 Multiple Person Recognition 4-14

4.4 Classification using raw data, Karhunen-Loève transform features, and identity network extracted features 4-15

4.5 Code Development 4-17

4.6 Summary 4-17

V. Conclusions 5-1

5.1 General 5-1

5.2 Multiple Day Verification 5-1

5.3 Identity Networks 5-2

5.4 Multiple Class Recognition 5-3

5.5 Neural Network Extracted Features versus KLT 5-3

Appendix A. Source Code A-1

A.1 Makefile A-2

A.2 dkmain.c A-3

A.3 backpropx.c A-12

A.4 dkiox.c A-15

A.5 ps.c A-25

A.6 psx.c A-34

Page

A.7 utils.c A-39

A.8 display.c A-45

A.9 globals_h A-46

A.10 macros.h A-46

A.11 globals.h A-48

A.12 makedata.c A-49

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure Page

2.1. Oja PCA Network 2-6

2.2. APEX Network 2-7

2.3. 5–Layer Network 2-11

2.4. 3–Layer Network 2-11

2.5. Network Architecture for Karhunen-Loéve Feature Extractor. (37) 2-13

2.6. Cottrell Neural Network for Face Recognition 2-16

2.7. Holons derived by PCA from hidden unit responses. (5) 2-17

2.8. Los Alamos National Laboratory Setup 2-18

3.1. 4-Layer Feature Extraction/Classification Network 3-3

3.2. Image compression, feature extraction and reconstruction 3-4

4.1. Non-smooth Identity Network Learning 4-2

4.2. Slow Identity Network Learning 4-3

4.3. Identity Network Learning, Eta = Varying 4-4

4.4. Identity Network: Input images versus reconstructed output images 4-5

4.5. Original, Scaled, and Shifted Image 4-12

4.6. Identity Network: Input images versus reconstructed output images using Motion Segmentation 4-16

List of Tables

Table Page

4.1. Average MSE of an Identity Network 4-3

4.2. Classification Accuracy Using Identity Network Extracted Features as inputs to a Cottrell (no hidden layer) backpropagation network 4-6

4.3. Classification Accuracy Using Identity Network Extracted Features as inputs to a single hidden layer backpropagation network 4-7

4.4. Classification Accuracy Using Identity Network Extracted Features of Multiple Day Data 4-8

4.5. Classification Accuracies for Single day verification of 3 target subjects 4-9

4.6. Classification Accuracies for Multiple day verification of 3 target subjects 4-10

4.7. Classification accuracies of original images versus scaled and shifted versions of original image 4-11

4.8. Classification Accuracies for Multiple Day Verification using data sets composed of 50% target/50% nontarget images 4-13

4.9. False Acceptance Testing of Multiple Day Data 4-14

4.10. Classification Accuracies for Multiple Classes using raw image data as inputs . . . 4-15

4.11. Classification Accuracies for raw image data versus Karhunen-Loève transform extracted features versus identity network extracted features. 4-18

AFIT/GE/ENG/92D-23

Abstract

The purpose of this study was to investigate and implement a neural network for face verification and classification. The research concentrates on developing a neural network based feature extractor and/or classifier which can be used for authorized user verification in a realistic work environment. Performance criteria such as recognition accuracy, system assumptions, training time, and execution time were analyzed to determine the feasibility of a neural network approach. Specifically, data was collected using a camcorder with automatic intensity compensation. Additionally, two segmentation schemes were used for data collection: manual segmentation and motion-based, automatic segmentation. The data consisted of over 2000, 32x32 pixel, 8 bit gray scale images of 52 subjects where each subject had from two to ten days worth of data collected. The data base was then used to create a number of training and test sets that varied by class size, training set composition, number of images per class, and so on. The training and test sets were then used to train and test the classification accuracy of the following networks: a back propagation network using the raw data as inputs; a back propagation network using Karhunen-Loève Transform coefficients, computed from the raw data, as inputs; and a back propagation network using features extracted by an identity network as inputs.

The results of the various network tests indicate that identity network extracted features performed well for single day captured data and poorly on multiple day data. Classification using the various networks, in general, performed well on constrained, single day captured, data bases but performed poorly when using data gathered over multiple days . For the multiple day problem, a verification network using a single hidden layer with backpropagation performed very well and was found to be most suitability for use in a face verification system.

FACE RECOGNITION WITH NEURAL NETWORKS

I. Problem Description

1.1 Introduction

Autonomous face recognition is the process of locating and identifying faces in a scene using pattern recognition techniques. While humans recognize faces many times a day with apparent ease, automating this process has challenged researchers for the past two decades. What does an automated face recognition system offer us to warrant the years of research this problem has received?

A system that automatically recognizes faces would be useful for several reasons. From a security perspective, an automatic face recognition system could enhance current access control systems by authenticating a users identity (32). Examples of such access control systems are secure computer systems, bank automatic teller machines, and automatic card readers. In fact, any organization or system that permits access based on a person's identity would find a face recognition system useful (32). Other security applications for a face recognition system would be criminal identification and scanning airports for terrorists. Finally, this system could be adapted for use in a speech recognition system or a visual communications system (32).

The remainder of this problem description begins with a background review of face recognition as it relates to pattern recognition and this thesis. ɪhis is followed by the problem statement, research objectives, assumptions, scope and limitations, and standards. The approach to the problem is then discussed and the chapter concludes with an overview of the remaining chapters.

1.2 Background

This section briefly discusses the pattern recognition process and then highlights some of the current research in the area of autonomous face recognition. The research highlighted below will be discussed in detail in chapter two but is included here for completeness.

Traditionally, pattern recognition is broken down into three areas: segmentation, feature extraction, and classification (38). Segmentation is the first step; it is finding regions of possible signals. The second step is feature extraction and in this step we search for the most important or significant features of the regions passed by the segmentor which can be used in the final step, classification. Classification compares the extracted features to those of previously identified objects and identifies the object as one of the previously identified classes. For this research the regions we desire to segment and identify are faces but the approach is the same whether we are using faces, trucks, tanks, jeeps, or words in speech.

1.2.1 **Turk and Pentland at the MIT Media Lab.** Matthew Turk and Alex Pentland, from the Massachusetts Institute of Technology Media Lab, have implemented an autonomous face recognition system that also makes use of the Karhunen-Loève Transform (40). While their system performs well, it has limited application because of the enormous amount of computational power necessary to operate the system—at present, the system requires three dedicated, high speed processors.

1.2.2 **Face Recognition at AFIT.** Face recognition at AFIT began in 1985 with early systems that were slow and required a significant amount of human intervention (31, 35). Through the years, various improvements were added and the system has evolved into what it is today (18, 33, 27). The current system is similar to the MIT system being developed by Turk and Pentland (36). However, the AFIT system has one major improvement over the MIT system. That is, the current AFIT preprocessor incorporates a centering routine and a gaussian window routine. These routines center the image and draw a gaussian window around the image, thereby de-emphasizing the background. With this improvement, the current AFIT system is 95 percent successful at identifying 55 face images (36). The current AFIT system is used as a benchmark for this research effort.

1.2.3 **Cottrell at UCSD.** Garrison Cottrell of the University of California at San Diego has been developing a neural network model approach to face recognition (11). His model is trained in a two step approach that uses an identity network for feature extraction and a single layer backpropagation network for classification. He has shown this network to be successful on limited databases.

1.2.4 **Los Alamos National Laboratory.** Another neural network approach to face recognition is being pursued at the Los Alamos National Laboratory (25). The lab uses a standard one-hidden layer network trained by backpropagation using raw image data as inputs. Currently, the network is limited to two class problems on a constrained data base but the preliminary results are very respectable.

1.3 Problem Statement

This research effort will focus on improving the classification performance and speed of the autonomous face recognition system by implementing the feature extraction and classification phases cf the recognizer in a neural network. The segmentation phase of the recognizer is being researched in a collateral thesis by Captain Kevin Gay (13).

1.4 Research Objectives

The objective of this research is to improve the algorithm for feature extraction and classification of faces in an autonomous face recognition system. Specifically, four network configurations will be developed and compared:

- An identity network for data compression and feature extraction,
- A back propagation network using the raw data as inputs,
- A back propagation network using Karhunen-Loève Transform coefficients, computed from the raw data, as inputs,
- And 3–layer and 4–layer feature extraction and classification networks using back propagation and raw data inputs.

1.5 Assumptions

- The distance from the subject to the camcorder is constant, with the exception of normal head movement.
- A friendly user is assumed, ie. the head orientation is face front.
- The network is expected to run on a Sun workstation.
- If using automatic segmentation ...
  - the face has been properly segmented from the background.
  - preprocessing to center and gaussian window the face has been accomplished.
- If using manual segmentation ...

- the face has been manually centered in the image.

- preprocessing to gaussian window the face has been accomplished.

1.6 Scope and Limitations

The scope of this thesis is to investigate the limitations of several new algorithms used for autonomous face recognition. All conclusions are based on test data.

1.7 Standards

The performance criteria for the algorithms are classification accuracy, user interaction, and modularity. However, accuracy is the most important of these criteria.

1.8 Approach/Methodology

A software environment will be developed and executed on a Sun SPARCstation2 that combines existing software with new software written in ANSI C. All algorithms will be developed with modularity as a key consideration. Data for training and testing will be gathered under as many varying conditions as possible to test the robustness of the algorithms.

1.9 Overview

Chapter Two presents a review of current literature related to face recognition systems with primary emphasis on feature extraction and classification. Chapter Three provides a detailed description of the methodology used in this thesis, and Chapter Four provides test criteria and results. Chapter Five presents conclusions based on the test results and makes recommendations for future study.

II. Literature Review

2.1 Introduction

This review examines some of the current literature in the area of autonomous face recognition. Face recognition research over the past several years falls into two categories: recognition using features and recognition using the whole face or holistic approach.

Face recognition using features was first attempted by L. D. Harmon in the early 1970s. He extracted features from profiles to identify the faces. His features were defined as the distance from the tip of the nose to the mouth, the distance from the nose to the chin, the distance from the eyes to the nose and other similar measurements (16). In addition to Harmon's method, other types of face recognition using features involves segmenting a face and then extracting features from the segments. Whatever the method, face recognition using features continues today with researchers all over the world (42, 2). The second category of face recognition is the holistic approach. Research in face recognition has moved towards a holistic point of view with researchers at the Massachusetts Institute of Technology (MIT) (40, 39), the University of California San Diego (UCSD) (11, 5) and AFIT (36, 15). The holistic approach still involves extracting features, but the features, which are extracted using some type of principal component analysis, are now taken from the entire face image, not just segments or profiles. This research is based on the holistic approach and it is what will be discussed in the following sections.

The remainder of this review begins with some brief biological notes on human face recognition. This is followed by a discussion of feature extraction and data compression using principal component analysis and neural networks. The review continues with discussions on several networks used for face recognition which include the Cottrell network, a standard multiple layer perceptron using backpropagation, and the Tarr/Ruck network for Karhunen-Loève transformations and classification.

Finally, several anecdotes are discussed that, while not directly related to this research, are still part of the face recognition literature and should be mentioned.

2.2 The Biology of Face Recognition

How does the brain accomplish face recognition? This is still a mystery but we are learning more each day. Experts believe that the biological process of face recognition occurs on the underside of both hemispheres of the brain in the temporal and occipital lobes (14). In other words, this process is localized (found in a specific area) in our brains. This theory is supported by the work of Rolls, Baylis, Hasselmo and Nalwa in their study of the response of specific neurons to faces (28). They tested a group of face responsive neurons in macaque monkeys and recorded the neuronal responses (action potential spikes per second) to a set of face stimuli and non-face stimuli. The results of their study indicate that there are neurons that react primarily to faces.(28).

Localization is also supported by the work of J. C. Meadows and A. R. Damasio in their studies of individuals who have lost the ability to recognize faces, a condition known as prosopagnosia. Both researchers agree that patients with prosopagnosia, when they have come to autopsy, always have bilateral lesions in the occipito-temporal regions of their brains (8, 20). Conversely, patients with bilateral lesions in other portions of the brain (ie., occipito-parietal region) do not have prosopagnosia. This also supports, albeit in a limited manner, the notion that the process is localized. Accepting localization as fact, what information does this area of the brain require to recognize faces?

One theory, based on psychological experimentation, suggests that we store the information for faces in a global to local scheme (41). In this experiment subjects were first asked to *classify* images as faces or non-faces and the time to classify faceness was recorded. Then each subject was asked to *identify* familiar faces which were either distinctive/unusual or typical faces. It was found that faceness could be assessed very quickly while recognition took longer. Also, a distinctive face could be identified more quickly than a typical face. As such, it was proposed that the *difference* information (ie. , color of eyes, shape of nose, hair style, etc.) is used in the identification process and the path. Information that is the same (ie. , the general shape of a face, two eyes, two ears, etc.) is used at a higher level in the recognition process and is not stored for each particular face. This could be considered a global to local type process which fits our human experience with face recognition because we often say that a person looks like so and so except for their eyes or nose, etc. This global to local idea is also supported in the prosopagnosia studies. Individuals with prosopagnosia can still identify a face as a face, but they can not identify who the face belongs to. In fact, many prosopagnosia patients also have difficulties with recognition of other objects such as a particular make of car or truck. As with the faces, they can identify a car as a car but they can not tell what specific type of car it is (8). This implies that we process general information first and then get down to the specifics for the actual identification process, but what are the specifics?

Experts agree that something is stored or encoded in the face recognition neurons; however, they disagree as to what that something is. One theory is that the information for a particular face is stored in a grandmother cell (28). In other words, all the information for a particular person is encoded in a single cell (neuron) and when we find that cell we know who the person is. A second theory, supported by the work of Rolls and his colleagues is that the information for recognizing a face is stored in a coded ensemble of neurons and these codes are recomputed whenever a new face is added to the system (28). This theory was tested by recording the activity of face responsive neurons to a set of known faces and then adding a novel face to the set and recording the responses to this new set. The tests indicated that adding a new face alters the *steady state* response of some of the neurons in question for a short time and then they reach a new, statistically different, steady state (28). This implies that information for face recognition is stored in our face neurons and we know who an individual is based on the firing pattern of this group of neurons. This also suggests that the entire code is changed, albeit very quickly, each time a new face is added to the system.

In any case, the biological theories surrounding face recognition are as varied as the number of researchers. The accepted theory today is that there are specific face recognition neurons in our brains and the global (faceness) to local (identification) idea is gaining momentum. From a pattern recognition point of view, this global to local idea could be considered data compression, which is important to the design of a face recognition system.

2.3 Feature Extraction and Data Compression

Feature extraction and data compression are important problems in pattern recognition and image analysis. Many times the goal is to find a set of features that represent the data as closely as possible and compress the data at the same time. There are many approaches to this problem and one of the more well known is Principal Component Analysis (PCA). Likewise, this problem has been addressed in linear and nonlinear PCA neural networks and in autoassociation networks as an unsupervised learning task (24). Each of these will be discussed in the following sections.

2.3.1 **Principal Component Analysis (PCA).** In general terms, PCA is a statistical method used for extracting features from a set of data with high dimensionality. It is a solution to the curse of dimensionality problem found in pattern recognition (9). PCA is a linear, orthogonal transformation or projection of the data onto a new coordinate system where the axis are uncorrelated and the maximum variance of the original data is found in only a small number of coordinates (12). Dimensionality reduction is achieved in this space by taking the coordinates which have the maximum variance and leaving out the coordinates with minimum variance. In mathematical terms, the basis vectors of the new coordinate system are the eigenvectors of the covariance matrix of the data and the variances are the corresponding eigenvalues (40). So in terms of PCA, the best projection, in terms of mean squared error of reconstruction, from an M to an N dimensional space, where $M >> N$, is then the N dimensional space which represents the N eigenvectors with the largest eigenvalues.

Researchers at the MIT Media Lab and AFIT use principal component analysis in their holistic approach to face recognition because it is believed that specific features, such as eyes or nose, may not be as important as the overall pattern of the face when it comes to recognition. A detailed review of PCA using the Karhunen-Loéve transform can be found in a masters thesis by Pedro Suarez (36). This approach is supported by the physiology and psychology of the face recognition process (40, 41). While the above process is mathematically sound, calculation of eigenvectors and projection coefficients is computationally expensive. As such, the MIT face recognition system is run using three computers: a Datacube Image Processor, a Sun 3/160, and a Sun Sparcstation. This system can perform the recognition task at a rate of two or three times a second (40) but the system size makes it impractical for current applications. Like the MIT system, the AFIT system is hosted on multiple computers. The preprocessing software is hosted on a NEXT computer and a Silicon Graphics computer, and the remainder of the software runs on a second Silicon Graphics computer. However, even with multiple computers, the process still takes several minutes to run. The question becomes how do we improve the speed of the system without giving up any of the accuracy? The answer may be a neural network approach to PCA.

2.3.2 **PCA using a Linear Neural Network.** When considering a neural network approach to PCA a starting point in much of the literature is the Oja algorithm (22). His network model, shown in

Figure 2.1, consists of a single linear neuron unit that uses a Hebbian type of learning rule. He showed that if the input vectors are a stochastic process this network tends to extract the largest principal component from the input vectors. This corresponds to the eigenvector in the covariance matrix that has the largest eigenvalue.

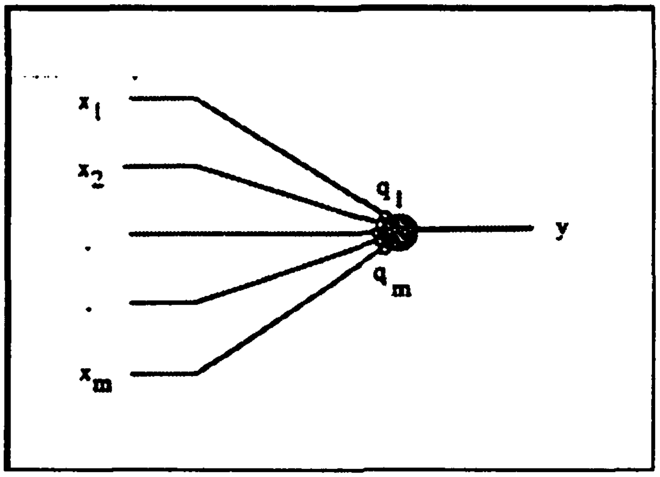

Figure 2.1. Oja linear PCA network which converges to the largest principal component of a stationary input sequence (12).

The output of Oja's net, y, and the learning rule for updating the weights, q_i, are

$$y = \sum_{i=1}^{m} q_i x_i \tag{2.1}$$

$$\Delta q_i = \beta(x_i y - q_i y^2) \tag{2.2}$$

where $x_i y$ is the Hebbian term that strengthens the connections when the input and the output are correlated. The second term, $-q_i y^2$, is used to prevent instability and makes $\sum q_j^2$ approach 1. Training the net in this manner maximizes the variance of the output given the constraint that $\sum q_j^2 = 1$. The disadvantage of the network is that it will only find the first principal component of the data set (12) and in many cases more principal components are required in order to be useful for a given problem.

The research continued and several algorithms have been developed that find multiple principal components of a set of data; one of the more recent algorithms is the Adaptive Principal-component EXtractor (APEX) (17). APEX was proposed by Kung and Diamantaras in 1990 and is stated to be recursive and adaptive in that, given the first $m-1$ principal components, it will find the m^{th} component. Additionally, they show that the m^{th} component is the largest component which is orthogonal to the previous $m-1$ components. Their network, shown in Figure 2.2, combines the properties of Oja's linear PCA net with a decorrelation scheme proposed by Foldiak in (12) that causes the network to work as a whitening filter.

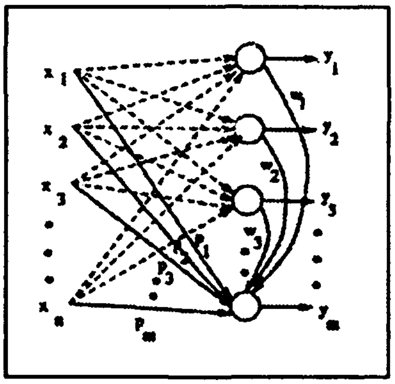

Figure 2.2. APEX network: solid lines denote weights p_i, w_j and are trained at the m^{th} stage. (Note that w_j asymptotically approach zero as the network converges) (17).

The outputs of the network are

$$y = Px \tag{2.3}$$

$$y_m = px + wy \tag{2.4}$$

where x is the input vector, y is the output vector representing the first $m-1$ output components, P is the matrix of $p_{i,j}$ weights of the first $m-1$ components, and p is the row vector of the $p_{m,j}$ weights of the m_{th} output.

The equations for the m_{th} component then become

$$\Delta p = \beta(y_m x^T - y_m^2 p) \tag{2.5}$$

$$\Delta w = -\gamma(y_m y^T + y_m^2 w) \tag{2.6}$$

where β and γ are positive learning rates.

Finally, if the above equations are expanded for each individual weight, the resulting equations are

$$\Delta p_{i,j} = \beta(y_m x_j - y_m^2 p_{m,j}), j = 1 \ldots n \tag{2.7}$$

$$\Delta w_j = -\gamma(y_m y_j + y_m^2 w_j), j = 1 \ldots m-1. \tag{2.8}$$

A review of these equations shows that equation 2.5 is simply Oja's hebbian update rule which was shown in (17) to force the outputs to the dominant principal components, and equation 2.6 is an anti-hebbian rule which was shown to cause the m_{th} output to be orthogonal to (or uncorrelated from) the previous $m-1$ components. Results of tests of APEX indicate that the principal components it extracts are almost perfectly normalized and orthogonal to one another and are very close to the actual components found using statistical PCA (17).

In addition to APEX, other algorithms for use in linear neural networks have been proposed for PCA. Sanger (34) proposed an algorithm that uses non-local information which complicates the analysis and Foldiak (12), mentioned earlier produces a set of vectors that span the same space as the principal components but are not the exact principal components (1). Additionally, an algorithm based on Successive Application of Modified Hebbian (SAMH) learning was proposed recently (1) which is shown to extract the principal components in an adaptive manner, similar to APEX, but is claimed to converge much more quickly.

PCA in linear networks is also supported by the work of Baldi and Hornik who showed that the error surface for this linear type network has a unique minimum that corresponds to the projection onto the subspace generated by the principal conponents of the input data set (3). However, they also state that the optimum solution using principal components could also be obtained using other well-known algorithms for computing eigenvalues and eigenvectors, and by numerical analysis standards, these algorithms are superior to using a linear neural network to extract principal components (3). Likewise, given data sets containing outliers, the linear networks have been shown to degenerate (19). With this in mind, why not disregard the notion of back propagation neural networks for feature extraction altogether? The answer: in addition to the simplicity of error back propagation, it can be applied to nonlinear networks, discussed in the next section, which have been shown to handle the outlier problem (19, 6). Additionally, it is very successful in a variety of other problems where there is no a priori knowledge of the structure of the mathematical properties of the ideal solution (3).

2.3.3 **Feature Extraction and Data Compression in Nonlinear Networks** In applications like pattern recognition and speech recognition an important problem is to find the relevant features in order to compress the data and still allow for correct classification or representation of the data (23). A reasonable requirement in data compression is that the original pattern can be restored from the compressed data with some acceptable level of error. This implies that feature extraction is data driven and independent of any pattern and in terms of neural networks, this is an unsupervised or selfsupervised learning task (23). One approach to the feature extraction problem has been the use of nonlinear autoassociation neural networks which have well-known approximation properties, can be expected to improve performance (23) and can solve the encoding problems where strict principal component analysis is degenerate (6). These networks are generally feedforward 3–layer, Figure 2.4, or 5–layer, Figure 2.3.

The 5–layer network of Figure 2.3 is believed to be more robust because it can theoretically compute any continuous mapping from inputs to the second hidden layer and another mapping from the second hidden layer to the output. The first and third layers are nonlinear (usually sigmoidal) and layers one, three, and five are linear. The output of layer three (p units), can now be used as the extracted features which are inputs to a classifier. The mapping from layer one to two can be considered a nonlinear PCA, and the mapping from two to three can be considered a linear PCA. In this case the dimensionality was reduced from n to p. However, a *significant* problem with this network is that the number of units in the nonlinear layers must be large, $N >> n$, in order to get good approximation capabilities for reconstruction (23). This constraint can be a problem for any network when training time is considered, especially if the number of inputs is already large[1].

An alternative to the 5–layer network is the nonlinear 3–layer network of Figure 2.4 which is also one solution to the training time problem of a 5–layer network. The 3–layer nonlinear network has hidden units that develop weight vectors that are believed to span the principal subspace of the input vectors, ie. they develop a distributed representation of the principal components (7). This result was found empirically by presenting the eigenvectors of the input images as inputs to the network (6). In

[1] For example, assume the input image is 32x32 (1024 pixels), the second layer and fourth layers of net are also 1024 nodes (not larger as is suggested), and the third layer is only 40 nodes, this network is estimated, conservatively, to require 80 days to train (based on current run times for similar networks on Sun workstations, see chapter 4).

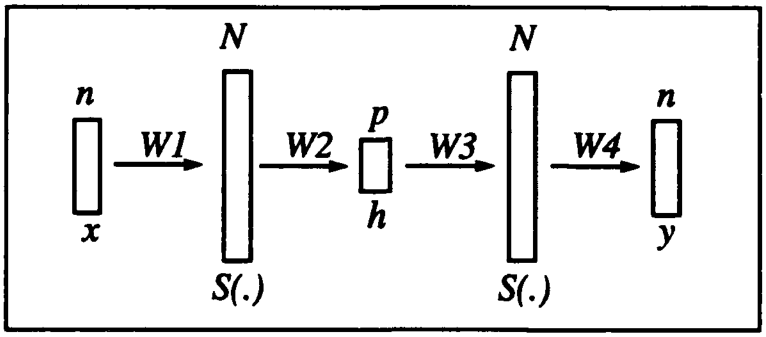

Figure 2.3. 5–layer network with linear and nonlinear layers. The number of units in a layer is given above the box, with $N >> n > p$, and the output is given below the box. (23).

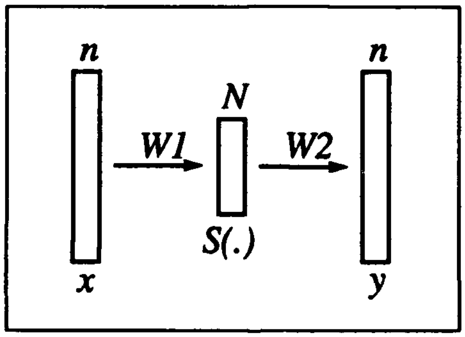

Figure 2.4. 3–layer network with nonlinear hidden and output layers. The number of units in a layer is given above the box and the output is given below the box.

other words, if the network is treated as a matrix M, where M represents the learned weights of the input layer, then find E' such that

$$E' = ME \tag{2.9}$$

where E is the matrix formed by the $k = N$, Karhunen-Loève generated eigenvectors of the input image in descending eigenvalue order as the columns. If the network spans the principal subspace of dimension k, then E and E' are related by

$$I' = E'E^T \tag{2.10}$$

where I' is the identity matrix. The results of this technique show a noisy first component with the rest of the components being slightly shortened (ie. the diagonal entries of I' are slightly less than 1) (6). It is believed that the hidden units develop a distributed representation of the principal components of the input data because the variances of the output activations of the hidden units are distributed more evenly than the variances of the components in the strict Karhunen-Loève transform (6).

It has been suggested that reducing the dimensionality with a 3–layer network is no better than using the standard Karhunen-Loève transform (21), however, this net has been used for image compression with good results (23, 5, 6, 7) and it is believed that the advantages of the nonlinearity will come out when the problem is nonlinear (6).

2.3.4 **Karhunen-Loève Network for Feature Reduction** A network was developed by Gregory Tarr and others at the Air Force Institute of Technology that performs feature reduction using a variation on the Karhunen-Loève transform (37). The network, shown in Figure 2.5, is described in detail in (37) but the idea is as follows. Every random vector, X, in a data set can be represented by a linear transformation of X with a matrix, A, such that

$$X = AY \tag{2.11}$$

where A is composed of the normalized eigenvectors of the data, X, covariance matrix in descending eigenvalue order and A does a rotation in the vector space of X to the new vector Y.

If the columns of A are orthonormal and form n linearly independent basis vectors, then the following conditions

$$A^T A = I \; and \; A^{-1} = A^T \tag{2.12}$$

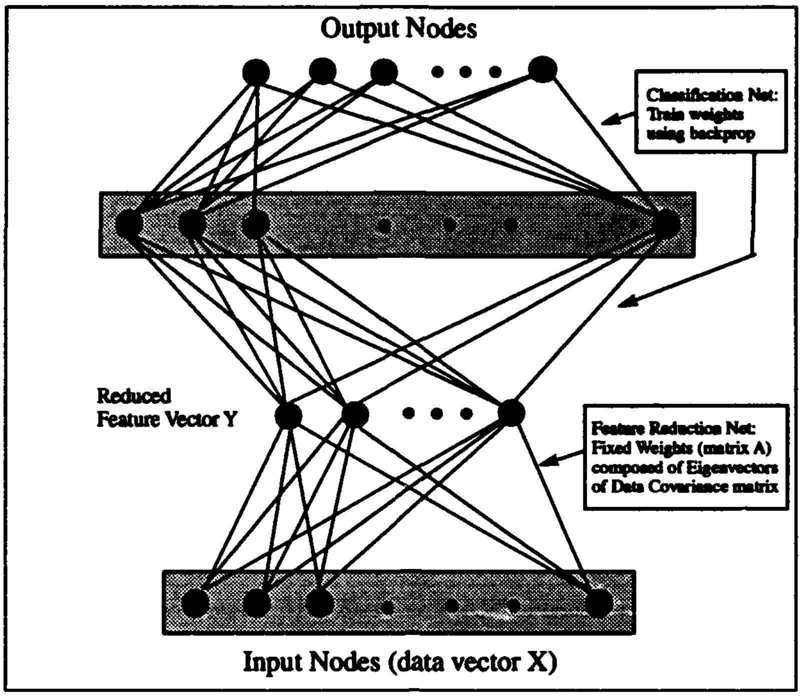

Figure 2.5. Network Architecture for Karhunen-Loéve Feature Extractor. (37)

allow the vector Y to be written as

$$Y = A^T X \tag{2.13}$$

where the dimensionality of Y is equal to the dimensionality of X, and Y is the vector of Karhunen-Loève coefficients found from the rotation of X to Y. If the goal is to reduce the dimensionality of Y, and if for all X presented to the network, some of the nodes of Y were very small, zero, or constant, then they would not be necessary to reproduce an estimate of X:

$$\hat{X}(m) = \sum_{i=1}^{m} y_i A_i + \sum_{i=m+1}^{n} b_i A_i \tag{2.14}$$

where b_i coefficients are independent of X and y_i coefficients are dependent on X (37). Now, if the objective is to classify and not reproduce X, then since the value $b_i, i = m+1 \ldots n$, is constant or nearly zero, it can be ignored. The number of eigenvectors chosen from A is then m and the network has effectively reduced the dimension of the data set. Likewise, since we are using PCA to reduce the dimension we have extracted the primary features which can now be used for classification of the data. The classification is performed by the top portion to the network, see Figure 2.5. It should also be noted that in order to make the network training easier the data must be statistically normalized before presentation to the feature reduction portion of the network and then statistically normalized once more before presentation to the classification portion of the network (37).

2.4 Neural Networks for Face Recognition

2.4.1 **Cottrell Neural Network.** A neural network for face recognition is being developed by Garrison Cottrell at the University of California at San Diego (11, 5). His network, shown in Figure 2.6, is trained using back propagation and is composed of his image compression (auto-association) network and a single layer perceptron. The parameters for the networks in Figure 2.6 are as follows:

- Auto-associator Network
  - Input nodes: 4096
  - Hidden nodes: 40, sigmoid activation range [-1,1]
  - Output nodes: 4096, sigmoid activation range [-1,1]
  - Momentum: 0
  - Hidden layer learning: 0.0001
  - Output layer learning: 0.1
- Classification Network

- Input nodes: 4096
- Hidden nodes: 40, sigmoid activation range [-1,1]
- Output nodes: 20, sigmoid activation range [-1,1]
- Momentum: 0
- Hidden layer learning: Fixed weights
- Output layer learning: 0.1

The basic operation of the network is as follows. First, the auto-associator network is trained to match output images to input images. The images for this experiment were 64x64 pixels, brightness normalized, and manually centered. There were eight images for each of 20 students, 10 male and 10 female, making a total of 160 images. The auto-associator network used for this phase consists of 4096 (64x64) input nodes, 40 hidden nodes, and 4096 output nodes. Successful training is defined as the root mean square pixel intensity error rate being less than 12 gray levels between input and output which corresponds to an average squared error per unit of .0017. The error rate is met in approximately 50 epochs which indicates that the network is trainable in a short amount of time. Dr. Cottrell theorizes that the hidden layer compresses the data and extracts features that represent a distributed representation of the principal components which are similar to the eigenfaces of the MIT face recognition system. He supports this theory by forming the covariance matrix of the hidden unit activations over all images and extracting the principal components using strict principal component analysis. The resulting principal components are then decompressed by running them through the output layer of the auto-association network. Figure 2.7 shows images reconstructed in this manner, and they are similar, at least in appearance, to the MIT eigenfaces.

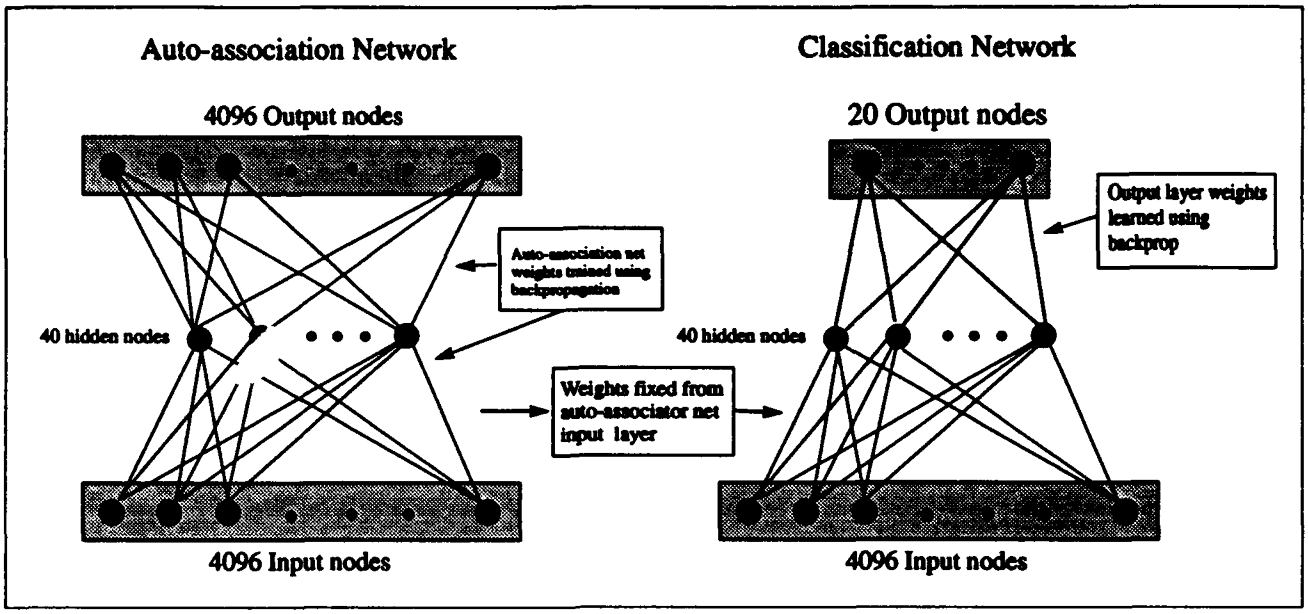
Auto-association Network
Classification Network
4096 Output nodes
20 Output nodes
Output layer weights learned using backprop
Auto-association net weights trained using backpropagation
40 hidden nodes
40 hidden nodes
Weights fixed from auto-associator net input layer
4096 Input nodes
4096 Input nodes

Figure 2.6. Left is the auto-association network used to train the input layer weights for the network on the right. The output layer weights on the classification network are trained using backprop. (5)

After the auto-associator network is trained, the weights between the input and hidden layer are fixed and the output of the hidden layer is connected to a smaller (20 output nodes), single-layer classification network. This smaller network is then trained to classify (identify) each image; the network is also trained to identify, and classify as 'unknown', nonface images at this time. The test results for this network were 99 percent recognition accuracy. However, the data base used was constrained to 20 subjects and all images were captured on the same day and time. The capabilities of this network have not been tested over multiple days of images or for larger data bases and it is believed that these tests will cause problems for the network.

2.4.2 **Backpropagation Neural Network for Face Verification.** Researchers at the Los Alamos National Laboratory have developed a face verification system that uses a 3–layer neural

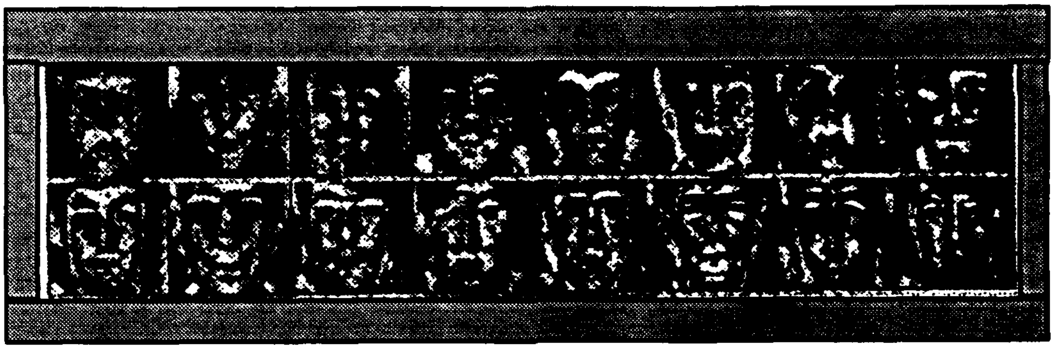

Figure 2.7. Holons derived by PCA from hidden unit responses. (5)

network trained by back propagation (25). The network parameters, which were arrived at empirically, are shown below:

- Input nodes: 1400
- Hidden nodes: 20, sigmoid activation range [0,1]
- Output nodes: 1, sigmoid activation range [0,1]
- Momentum: 0.50
- Hidden layer learning: 0.15
- Output layer learning: 0.30

The network is fully connected and all weights are initially set to random values uniformly distributed between −0.1 and 0.1. The input vectors are scaled to the range [-1,1] and the network was trained to output a 1.0 for the target and a 0.0 for outputs other than the target. The training was stopped after 15,000 iterations. In all cases the number of target pictures in the data base was artificially forced to be 10 percent of the total by replicating the target images.

The data base used contains 11,416 images of 760 different people with the number of images per person ranging from 5 to 20. All images were aquired using a video camera with VideoPix installed on a Sun Microsystems SparcStation IPC computer. As can be seen in Figure 2.8, the distance from the camera, background, and lighting conditions were kept as constant as possible.

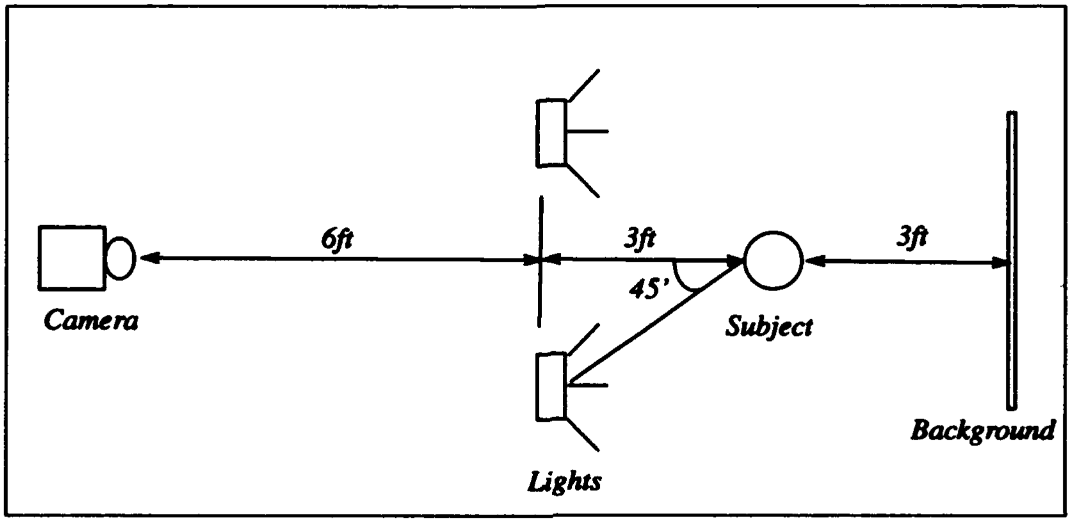

Figure 2.8. Setup used for capturing images at Los Alamos National Laboratory. Images are then used in a Backpropagation Network. (25)

The network was tested using various compositions of training data. The training sets consisted of 5, 10, 15, or 20 percent of the data base either randomly selected or from specific demographic groups. The results of the verification tests were averaged over all training scenarios and the numbers show a 99.997 percent correct rejections of non-targets and a 91.3 percent correct acceptances of targets. As in the Cottrell tests, the data used for this network testing was captured during a single sitting on one day. To be useful, a verification network must perform well on multiple days of data and this network is not expected to perform well on multiple days. Additionally, this network has not been tested for classes of data larger than one so the upper limits of the network are unknown. These concerns will be addressed in this research.

2.4.3 **Additional Network Research** An image recognition system which uses an infrared illumination system to overcome ambient illumination is currently being developed in England (10). The system is based on a neural network known as WISARD and is reported to be capable of face recognition against background scenes. The drawback of this system is the complexity: it requires two cameras, control hardware, an IR Illumination System, Training Software, and WISARD hardware. Likewise, it requires a significant amount of memory because two 800x540 images are taken for every scene.

In addition to WISARD based system, a neural network for face recognition based on a multilayer perceptron and shared weights has been introduced in France (4). It is believed that this sytem will handle changes in lighting and rotations and face translations by adding additional prototypes. Preliminary test results have yielded acceptable results but the database consisted of only 10 subjects. This system will also require large amounts of memory because the images used were 256x256 pixels, with 70 prototypes captured per person. This equates to approximately 640 Megabytes of data.

2.5 Summary

This chapter presented several neural network approaches to feature extraction, data compression, and classification. Specifically, in relation to face recognition, an identity network, a Cottrell classification network and a backpropagation network were discussed. Each was reported to perform well for face recognition, however, the data bases used were limited in size and/or in the number of classes. These limitations will be addressed in this research by increasing the size of the data bases and increasing the number of classes in the data bases. Additionally, the capabilities of these networks for data captured over multiple days has not been investigated. Therefore, this research will also focus on capturing data over multiple days to determine how this type of data effects the networks described previously. Chapter 3 presents the methodology used to address these issues.

III. Methodology

3.1 General

The objective of this thesis was to investigate various neural networks for face recognition to determine their capabilities as feature extractors and classifiers. Since one of the criteria was to implement the algorithms on Sun workstations, the first task was to port an existing multilayer perceptron algorithm onto a Sun workstation and test its operation. Next, the algorithm was enhanced to work in the following modes (refer to Appendix A for code):

- As a multilayer perceptron (MLP) with sigmoidal or symetrical sigmoidal activations
- As an identity network for image compression and feature extraction
- As a Cottrell classification network
- As a 4-layer feature extraction and classification network

After the MLP algorithm was ported and tested, the second task was to develop and test the identity network. Next, the problem of individual verification was examined. For this task, the algorithms were used to verify individuals in a two-class problem where class 1 signified a positive verification and class 2 an unknown or negative verification. Next, the individual verification problem was expanded to include multiple days. The multiple day problem is an important extension because it has not been investigated in the literature to date. After verification tests were complete, the algorithms were used in a large scale recognition problem. This differed from the verification task in that the network was now a multiclass problem with many individuals to recognize in the database and this is a much more difficult problem than verification. The databases used in both the verification and multiple class tests were compared using a standard multilayer perceptron and the neural network feature extractor/classifier. Additionally, the performance of a network using standard Karhunen-Loève Transform (KLT) coefficients as input features was compared to the neural network extracted features. The final task was to describe the algorithms developed to implement the above objectives.

3.2 Algorithm Development

The first phase was to port an existing MLP algorithm with back propagation learning to a Sun workstation. The MLP algorithm used for this phase allowed a maximum of three layers of nodes (input, hidden, and output) and used sigmoidal activations in the hidden layer and output layer. The sigmoids in this case had limits of [0,1]. The algorithm was then modified to function as both a Cottrell and an Oja identity network. For this, symmetrical sigmoid activation functions, limits of [-1,1], were added to the algorithm and were interchangeable with the current sigmoid activation functions. The difference between the Cottrell identity network and the Oja network is that the Oja network uses linear activations on the output layer. Additionally, the capability to display the network input image and the output image was added to allow for visual verification of the performance of the identity networks. The next enhancement to the algorithm was to allow it to perform as a Cottrell face classification network. This meant that the network algorithm had to initialize and learn the output layer weights but the input layer weights were those previously learned from an identity network algorithm. The final enhancement to the algorithm was to create a 4-layer network for feature extraction and classification. This was done by adding another layer to the Cottrell classification network. It is believed that this additional layer will allow for increased accuracy in the classification of data since it essentially adds a hidden layer to the classification portion of the network. It is simply a backpropagation network whose inputs are the outputs of the hidden layer of an identity network, see Figure 3.1.

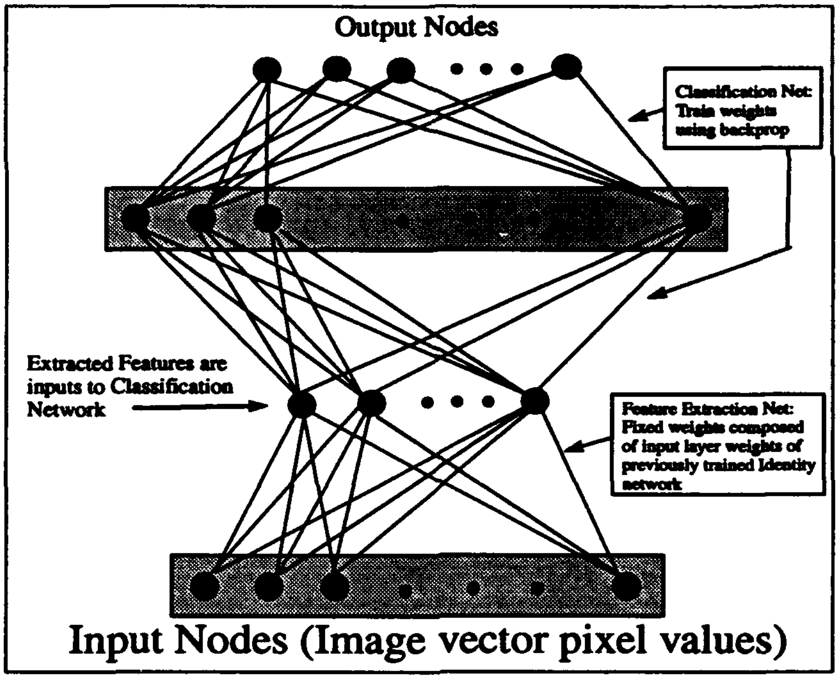

Figure 3.1. 4-Layer network for feature extraction and classification. The input layer consists of the input layer of a previously trained identity network and the upper layers consist of a multilayer perceptron with backpropagation learning.

3.3 Feature Extraction and Classification Using Identity Networks

The goal for this task was to determine whether or not an identity network, shown in Figure 3.2, could extract features that were meaningful as inputs to a classification network. The idea was to allow the identity network to learn a compressed representation of the data and reconstruct it.

The data for this task and subsequent tasks consisted of 32x32 (1024) pixel, 8 bit gray scale images of faces that had been captured on the Sun workstations using VideoPix. Each individual had between 10 and 20 images captured on any given day with a total of 800 images of 49 different individuals. All images were windowed with a gaussian window routine to deemphasize the background.

The network required for feature extraction consisted of 1024 input nodes and 1024 output nodes. The number of hidden nodes used determined the amount of compression, or the number of features extracted, and this number was varied from 10, to 20, to 40. 40 was used as an upper limit due to training time required. The network was trained until the averaged mean squared error per pixel (MSE/P) between the original image and the reco..structed image was less than .0034 which corresponds to 25 gray scale levels per pixel or 10 percent reconstruction error per pixel. The reconstruction in this manner is recognizeable to a human as will be shown in chapter 4. MSE/P was found using the following definition

$$MSE/P = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N}(IN_{ij} - OUT_{ij})^2}{(M)(N)} \tag{3.1}$$

where N is the image size in pixels, M is the number of images, and IN and OUT are, respectively, the input and output node values.

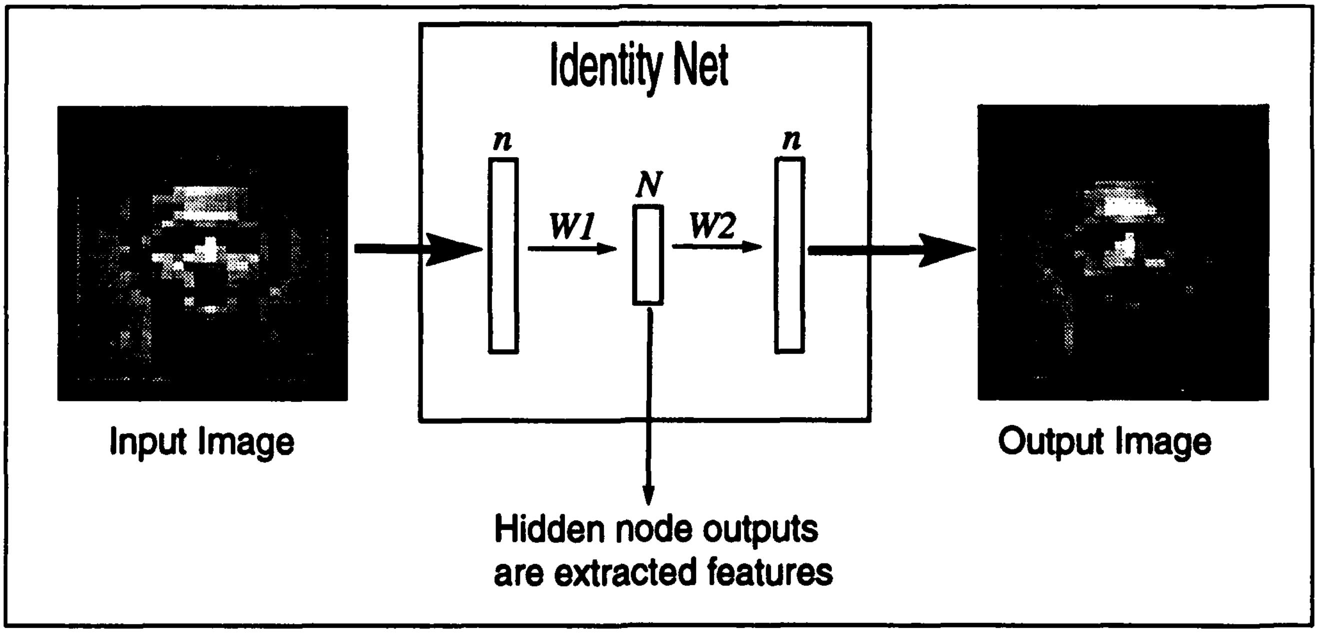

Figure 3.2. Image compression/feature extraction and reconstruction procedure: The outputs of the hidden nodes are the features used as inputs to a classification network. (11, 5).

In addition to varying the number of hidden nodes, the size of the database in terms of the number of classes was also varied to determine what effects, if any, this had on the networks. The databases used for these tests were varied from 10 classes, to 30 classes, to 49 classes.

3.4 Verification and Classification Using Backpropagation

This task is divided into two phases: single day verification and multiple day verification. The two differ in the data sets used for training and testing the network. Single day data are images that were captured on one day only within a 90 second window for each individual. Multiple day data are images that were captured over several days for each individual.

3.4.1 **Single Day** For this task several databases were generated that had the target individual to verify as class 1 and images of other subjects as class 2. Multiple runs were made of each database and the average accuracy (correct verification) was recorded. The minimum number of images used in any database was 100 to keep in line with the testing performed in (25). The mix of individuals in each data set was 20/80, which means that 20 percent of the images in the data set were of the target individual (class 1) and 80 percent were of nontargets (class 2). Nontargets consisted of 2 images each of other individuals in the data base.

The total number of images in each database was 50: 10 for the target (class 1) and 40 (2 images each of 20 other individuals) for the non-target (class 2). Each database was trained and tested using the following:

- A standard multilayer perceptron with nonlinear activations on the output layer and hidden layer,
- A Cottrell (single layer) classification network with Cottrell identity network extracted features as inputs,

- The 4-layer network which uses identity network extracted features as inputs to a classification network with a hidden layer.

This totaled 6 tests for each database. There were also additional tests to determine the numbers of hidden nodes and iterations which provided for the best performance.

3.4.2 **Multiple Day Verification** This task is one of the most significant contributions of this thesis because it has not received attention in the face recognition literature. It is the verification of individuals using databases that are generated over time. Most research consists of images that have been captured in a relative short period of time, one or two minutes, and under very strict conditions. This task was developed to test the system over less constrained input images because, in reality, we change from day to day. The questions are then, how does data from multiple days effect the accuracy and what are the limitations of this system? The approach was as follows:

1. Use the databases from the individual verification task to train the various networks.
2. Test each network using data captured from a new day.
3. After testing, add the new days data to the training database and retrain the network.
4. Go to step 2, and continue this process for the desired number of days.

3.4.3 **Multiple Person Recognition** For this experiment, the size of the database was increased as well as the number of classes. This is a more difficult problem than individual verification because we are now attempting to classify (recognize) many faces as opposed to just one face. The database for this task consisted of 800 images of 49 different individuals. Tests were accomplished on multiple class problems using 10 classes (10 different people), 30 classes, and 49 classes. As in earlier tests, each individual had 10 or 20 images captured and the data was randomly separated in each case to be 60 percent training data and 40 percent test data. The point of this part of the study was to determine what effects larger databases have on the networks being tested.

Several questions to be answered are listed below.

- What is the trade-off between the size of the database and classification accuracy?
- How many hidden nodes are required for a given number of classes?
- Do larger databases make training time prohibitive?

3.5 Neural Network Classification versus Karhunen-Loève Transform

For this part of the study the classification accuracy using the identity network extracted features was compared to classification accuracy using standard Karhunen-Loève Transform features (coefficients). The results for multiple person recognition using the Karhunen-Loève Transform are documented in a previous thesis by Pedro Suarez (36) and in a collateral thesis by Kenneth Runyon (30). Suarez showed that the Karhunen-Loève Transform, using single day data, obtained an accuracy of 95 percent on 55 faces (classes of data). Runyon used the Karhunen-Loève Transform on data gathered using a motion based segmentation system developed in a thesis by Kevin Gay (13). Runyon had data sets for single day, multiple classes and two day, multiple classes. Using the motion segmented data and the Karhunen-Loève Transform Runyon achieved accuracies of 77 percent for single day and 32 percent for two day data using 23 faces (classes of data). The testing in this section will be for a comparison of the Suarez and Runyon systems to the neural network system. As such, data for a multiple class, single day training and multiple class, two day training will be used for comparisons.

Additionally, the data sets used for single and two day testing in the neural networks will be tested in the Runyon system.

3.6 Code Development

All code for this thesis was written in ANSI standard C. Some routines were developed specifically for this thesis, some were borrowed from others and some were taken from Numerical Recipes In C (26). Code developed for this thesis is included in Appendix B.

3.7 Summary

This chapter presented the methodology for investigating various neural networks used for feature extraction and classification of face image data. Specifically, an identity network was developed and tested as a nonlinear neural network feature extractor. Likewise, a single layer (Cottrell) and a multilayer backpropagation network were developed for use as classification networks with identity network extracted features as inputs. Additionally, a backpropagation network for verification and recognition, using data gathered over multiple days, was developed and tested. In all cases, data bases used for testing and training were varied by the number of exemplars used, by the number of classes, and by the days on which the data was gathered. Test results for these networks are found in Chapter 4.

IV. Results

4.1 General

This chapter presents the test results for the tasks outlined in Chapter 3. Images used were 32x32, 8 bit gray scale that were captured using a Sun workstation tool called VideoPix. More information on VideoPix can be found in a collateral thesis (13). Images were preprocessed to add a gaussian window around the faces in order to de-emphasize the background.

4.2 Feature Extraction and Classification Using Identity Networks

4.2.1 **Feature Extraction** The image compression capabilities of the identity network was tested first because the outputs of the hidden layers in such a network will be the extracted features for use in a classification network. The network used, shown in Figure 2.4, had the following parameters which were based on the work of Cottrell (5) and Oja (23):

- Input nodes: 1024
- Hidden nodes: 10, 20, or 40, sigmoid activation range [-1,1]
- Output nodes: 1024, sigmoid activation range [-1,1]
- Momentum: 0.
- Hidden layer learning: 0.0001
- Output layer learning: 0.1

Training the network using these parameters, especially as the number of classes increased, caused the output error to begin bouncing slightly, see Figure 4.1, indicating that the learning rate used could be too large.

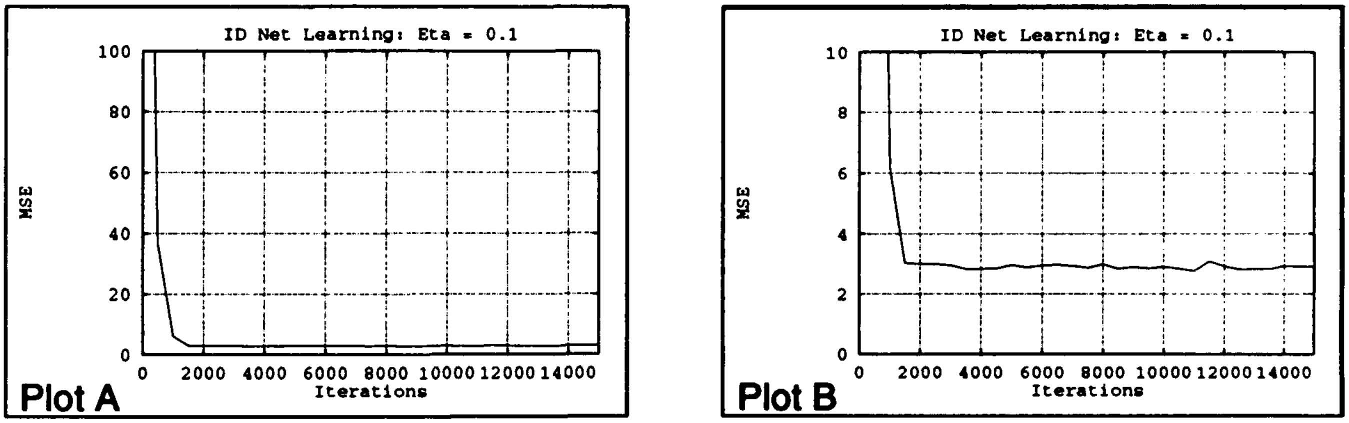

Figure 4.1. Non-smooth identity network learning. Plot B represents a more detailed look at Plot A. Non-smooth learning, refer to Plot B, can be seen after 4000 iterations, indicating that the learning rate is too large. In this case eta = 0.1

The solution was to retrain the network using a smaller output layer learning rate; 0.01 instead of the 0.1 of the original training. This solved the non-smooth behavior, see Figure 4.2, but the final MSE was now generally above the target range of 3.5 after the desired number of iterations. This problem was solved using the following variable learning rate: 0.25 for epoch number 1, 0.1 for epochs 2–25, and 0.01 for epochs 26 and higher. The increments of 25 epochs were chosen through trial and error. This variable learning provided for a smoother MSE curve, see Figure 4.3, and limited the training time of each network, run on Sun SPARCstation 2's, to 8 hours or less.

With the parameters now defined, the identity network performed as desired and the output of the network at various stages of learning is shown in Figure 4.4. As evidenced in the figure, the network can learn to compress and reconstruct, to an acceptable level, the images in a data set. Results of testing the network using various class sizes and hidden nodes is shown in Table 4.1.

The table lists the final average MSE for all images in the training set after 15,000 iterations. The results show that the target MSE of 3.5 or less was achieved for 10 and 30 classes but was slightly

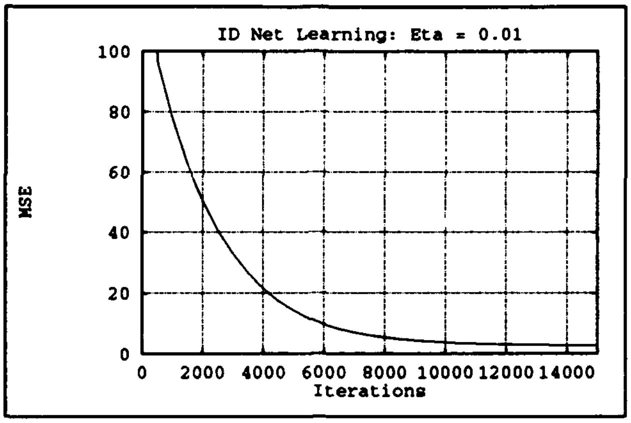

Figure 4.2. Slow identity network learning indicating that the learning rate is too small. In this case eta = 0.01

Table 4.1. Average MSE of an Identity Network after 15000 iterations for varying class sizes and numbers of hidden layer nodes. For the Identity Network, inputs = outputs = 1024.

	10 nodes	20 nodes	40 nodes
10 classes	2.7	1.6	0.9
30 classes	3.3	2.3	2.1
49 classes	5.3	4.0	7.3

higher for 49 classes. The 15, 000 iteration limit was placed on the networks to limit the training time and it is believed that the final error for 49 classes could be lowered given a much longer training time: this was tested with a single run where a final MSE of 3.8 was reached after 50, 000 iterations.

Finally, it should be noted that the input layer weights learned by each identity network were saved for use as the input layers of the classification networks because the output activations of this layer would be the features used in the classification networks.

4.2.2 **Classification Using Identity Network Extracted Features** With the identity networks trained, the next phase was to test how well features extracted from the identity networks worked in

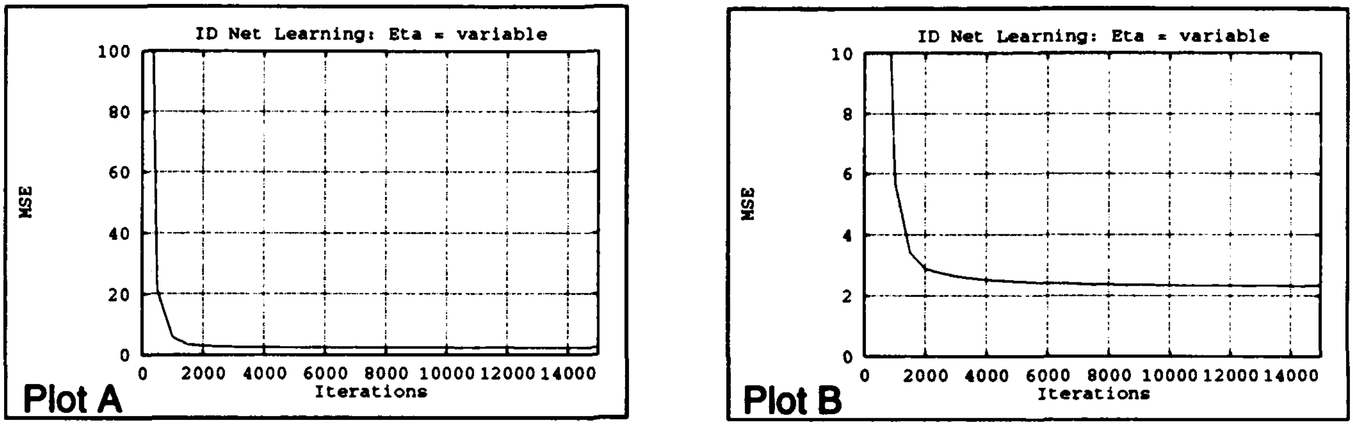

Figure 4.3. Identity network learning obtained with variable learning rate. The net learns quickly at first and more slowly and smoothly as the iterations increase. Plot B represents a more detailed look at Plot A.

a classification network. Recall that the extracted features used here are the output activations of the hidden layer nodes of an identity network, refer to Figure 3.2. For this test, the network used is that shown in Figure 3.1 where the input layer is the fixed weights from the previously trained identity networks and the output layers are a backpropagation network with the following parameters.

- Input nodes: 1024
- Hidden layer 1 nodes: Fixed by Identity net training at 10, 20, or 40
- Hidden layer 2 nodes: Varied from 0, for a Cottrell classifier, to 50
- Output nodes: Multiple classes of 10, 30, or 49
- Momentum: 0.50
- Hidden layer learning: 0.15
- Output layer learning: 0.30

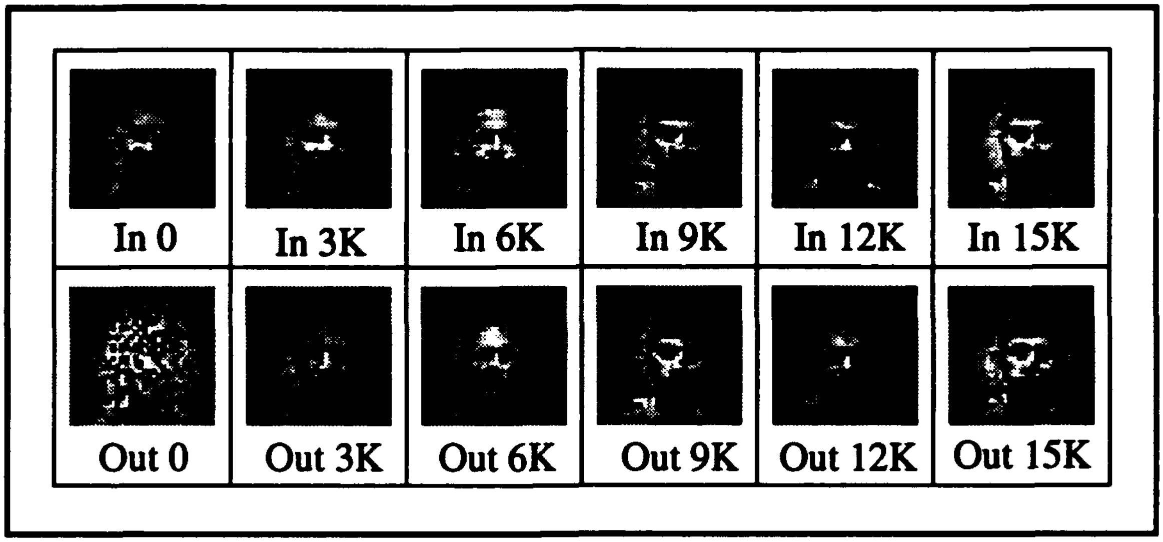

Figure 4.4. Identity network input images versus reconstructed output images at selected iterations of 0, 3000, 6000, 9000, 12000, and 15000.

- Activation Function: Symmetrical sigmoid, range [-1,1], at hidden layer 1 and sigmoid with range [0,1] at hidden layer 2 and the output.

The results of the various classification network configurations are summarized in Tables 4.2 and 4.3, which show that the features extracted from the identity networks are acceptable features to classify the data.

The results show that the feature extraction and classification networks used by Cottrell (a 100/1 compression or 10 hidden nodes in identity network and no hidden layers in the classification network) had a significant decrease in classification accuracy as the number of classes was increased. However, the accuracy was improved by decreasing the Cottrell recommended compression of 100/1 to 25/1 (40 hidden nodes in the identity nets). Additionally, adding a hidden layer of nodes to the classifier also improved the classification accuracy of the test set, refer to Table 4.3. As the table shows, varying

Table 4.2. Classification Accuracy Using Identity Network Extracted Features as inputs to a Cottrell (no hidden layer) backpropagation network. As the class size increases the Cottrell network using 10 input features performs poorly. The accuracy was greatly improved when the Cottrell network was modified to have 20 and 40 input features.

Cottrell Classifier Test Set Accuracies		
Input Features	Output Classes	Test set accuracy
10	10	87.5%
10	30	85.0%
10	49	66.0%
20	10	95.0%
20	30	97.0%
20	49	91.8%
40	10	85.0%
40	30	96.7%
40	49	97.9%

the number of hidden nodes (between 10 and 50) in the classification portion of the network had a slight impact on the test set accuracy; too few or too many nodes in the hidden layer caused the test set accuracy to decrease. The trade off then is between the accuracy desired and the training time involved for larger networks. As stated earlier, the identity nets with 40 hidden nodes required 8 hours of training time on Sun workstations; this must be added to the time to train the classification portion of each network, and this time grows as the number of hidden nodes is increased. However, given the trade offs, the bottom line here is that the features extracted using the identity networks can be used to classify the data with acceptable accuracy.

When comparing these results to those obtained by Cottrell (11, 5), a 100/1 compression ratio (10 hidden nodes) did not yield the 99 percent accuracies he reported. In fact, as the number of classes was increased to 49, which is over twice the 20 classes Cottrell used, the accuracy drops to a low of only 66 percent. However, by increasing the number of hidden nodes to 40, which increases the number

Table 4.3. Classification Accuracy for test data using Identity Network Extracted Features as inputs to a single hidden layer backpropagation network. This type of network is an improvement over the Cottrell classification network. Accuracies are listed with respect to the number of nodes in the hidden layer. All training sets obtained a 100% training accuracy during training.

Input Features	Output Classes	10 nodes	20 nodes	30 nodes	40 nodes	50 nodes
10	10	90.0%	90.0%	97.5%	92.5%	90.0%
10	30	80.8%	85.0%	89.1%	85.0%	85.0%
10	49	64.3%	71.4%	73.0%	73.0%	75.0%
20	10	95.0%	95.0%	95.0%	97.5%	95.0%
20	30	88.3%	91.6%	93.3%	94.1%	89.1%
20	49	77.0%	84.1%	89.7%	88.2%	86.2%
40	10	85.0%	95.0%	85.0%	85.0%	80.0%
40	30	92.5%	94.1%	96.7%	93.3%	91.7%
40	49	80.0%	91.8%	91.8%	91.3%	92.8%

of features (inputs) used in the classification network, the classification accuracy increased over 30 percent. Therefore, modifying the Cottrell identity network to a 25/1 compression ratio allows the network to perform well for class sizes as larger as 49. It is also important to note that for this research the input images were 32x32 pixels and Cottrell used images of 128x128 pixels; in other words, the modified identity network (ie. 25/1 compression) performs well and allows a 75 percent reduction in the amount of input data required.

4.2.3 **Identity Networks and Multiple Day Data Classification** For this test, data gathered on a second day was used to test the identity networks generalization capabilities for feature extraction. Specifically, the data from day 2 was input to the previously trained identity networks (trained using day 1 data) and the features for that data (activations of the hidden nodes) were saved. The features were then tested using the weights of the previously trained classification networks of Table 4.3. Results of this testing are shown in Table 4.4. These results show that the identity networks do not generalize well over multiple days of data. Likewise, adding additional days worth of data to the data sets did not improve the classification accuracy. This is because multiple days of data create problems for the identity networks; the identity network can minimize the MSE of the training set but when multiple days of data are used, the error on the test set remains a magnitude higher which means that the test set error, in terms of reconstruction, is plus or minus 100 percent. As such, the features extracted by the identity network for the test set are not adequate for proper classification.

Table 4.4. Classification Accuracy using Identity Network Extracted Features of multiple day data as inputs to a 2 weight layer backpropagation network for varying numbers of hidden layer nodes and a single weight layer Cottrell classification network. NOTE: 0 nodes represents the Cottrell network results.

Output Classes	Input Features	10 nodes	20 nodes	30 nodes	40 nodes	50 nodes	0 nodes
10	10	5.0%	5.0%	12.0%	10.0%	3.0%	10.0%
10	20	5.0%	8.0%	8.0%	10.0%	5.0%	10.0%
10	40	10.0%	19.0%	19.0%	20.0%	14.0%	10.0%
30	10	5.0%	5.0%	3.0%	1.0%	3.3%	3.3%
30	20	5.0%	6.7%	2.3%	4.3%	4.7%	3.3%
30	40	6.0%	4.7%	5.3%	5.3%	3.3%	3.3%

4.3 Verification and Classification Using Backpropagation

4.3.1 **Single Day Verification** The images used for this test had been captured during a single sitting of each subject over about 90 seconds. The training and test data sets were generated such that the target subject (for verification) was class 1 and all other faces in the data sets were class 2. The network used for this test had the following parameters:

- Input nodes: 1024

- Hidden layer nodes: 10, or 20

- Output nodes: 2
- Momentum: 0.50
- Hidden layer learning: 0.15
- Output layer learning: 0.30
- Activation Functions: Sigmoidal (range [0,1]) at hidden and output layers.

The goal was to affirm that that the network could correctly verify a target individual. Test results, representing 10 runs each for 3 targets, are shown in Table 4.5. The results show that the network performed as expected and can verify individual targets using a standard backpropagation network.

Table 4.5. Classification Accuracies for Single day verification of 3 target subjects

Data (day)	Hidden layer nodes	30 run average
day 1 vs day 1	10	96.4%
day 1 vs day 1	20	96.9%

These results support the verification work being performed in Los Alamos National Laboratory (25) which uses raw image data as inputs to a backpropagation network with a single hidden layer. The Los Alamos Laboratory research uses a minimum of 100 images to train the network, with 10 percent of the images representing the target to verify and 90 percent representing nontargets. Additionally, their data was captured on a single day for each individual. Data captured in this manner yields good results (Los Alamos had results as high as 99 percent for some individuals) but it does not allow for daily changes in individuals and, therefore, does not represent the real world where images will have to be captured on many days if the system is to be useable for verification. The multiple day day problem is addressed in the next section.

4.3.2 **Multiple Day Verification** The accuracy for verification using data captured on a single day is very good. However, it is believed that training on a single days data will not allow for verification in a real environment where images will be gathered daily. This sections testing was designed to find the problems of a multiple day system and propose and test solutions. The network used for testing in this section is the same as that for single day verification, ie. single hidden layer backpropagation network, but the difference is in the data sets used for training and testing. Data used in this section was gathered over several days with multiple sittings of each target. The results of the testing are found in Table 4.6.

Table 4.6. Classification Accuracies for Multiple day verification of 3 target subjects using raw image data as inputs to a single hidden layer backpropagation network. All training sets obtained a 100% classification accuracy during training.

Data (day)	Hidden layer nodes	30 run average
day 1 vs day 1	10	97.0%
day 1 vs day 1	20	96.6%
day 1 vs day 1	40	95.3%
day 1 vs day 2	10	11.7%
day 1 vs day 2	20	15.0%
day 1 vs day 2	40	20.5%
day 1–2 vs 1–2	10	91.1%
day 1–2 vs 1–2	20	90.8%
day 1–2 vs 1–2	40	89.0%
day 1–2 vs 3	10	9.0%
day 1–2 vs 3	20	15.0%
day 1–2 vs 3	40	2.0%
day 1–3 vs 1–3	10	90.6%
day 1–3 vs 1–3	20	90.7%
day 1–3 vs 1–3	40	90.3%
day 1–3 vs 4	10	14.0%
day 1–3 vs 4	20	33.0%
day 1–3 vs 4	40	44.0%

Column one indicates which day(s) the training versus test sets were captured on. Column three shows the classification accuracies for the various data sets and hidden node configurations. As expected, the network performed poorly in the multiple day test, but it was proposed that increasing the number of days in the training set (ie. more prototypes) would show an improvement. However, as can be seen in the table, adding prototypes from several days provided no improvement over the single day training. Therefore, a test was developed to determine why the network was not improving even with the introduction of multiple days images into the training set.

For this test a training data set was created that contained target images from a single day only and the network was trained. Then a test set was created by shifting these images 1 pixel at a time and testing the net to determine how a shift in the image would effect the verification accuracy. Likewise, the images were scaled to determine if scale was a problem. An original image with a shifted and scaled version are shown in Figure 4.5 and the results of the testing are provided in Table 4.7. The results of these test indicate that shifts of 1 pixel can be overcome by some network configurations since the accuracy for a 1 pixel shift varied from 0 to 100%. However, if the shift is 2 pixels, or about a 6.6% shift, then the network performs poorly. Likewise, a scale change of even 5 percent will cause severe problems for the network.

Table 4.7. Classification accuracies of original (day 1) images versus scaled and shifted versions of the same images. Original images were trained to 100% accuracy of the training set; original image test set accuracies are shown in the table.

Test Set Classification Accuracy: 10 run average, 10 test images per run			
Original Images	Shifted 1 pixel	Shifted 2 pixels	Scaled 5%
99.0%	40.0%	00.0%	03.0%

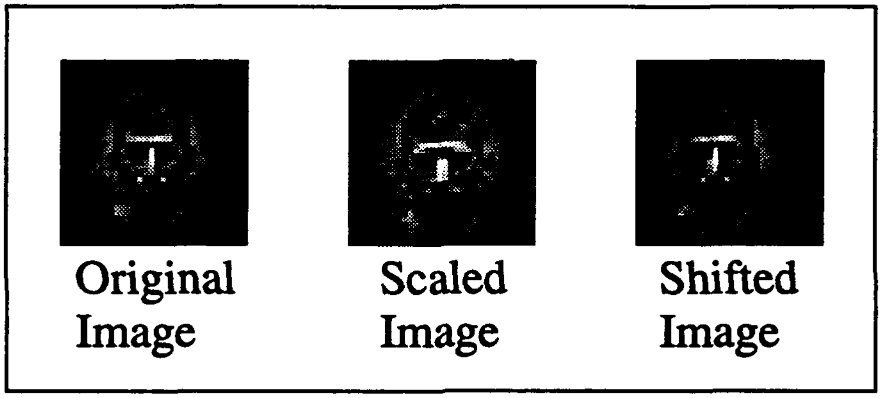

Figure 4.5. Original image and a scaled and shifted version of the same for testing the effects of scale and shift on the multiple day verification networks

The question was then how do you compensate for the scale and shift problems? More prototypes still seemed to be a piece of the solution but that alone was shown previously to offer little improvement. The proposed solution was to increase the number of days of training data plus increase the mixture of target versus nontarget images in the training set. This means that instead of using a data set with 20 percent of the images from the target and 80 percent nontarget, a new data set containing a 50/50 mix would be created and tested. It was believed that this would allow the weights to update more evenly for target and non target alike.

The networks were retrained and the results for 2 target subjects are shown in Table 4.8. As can be seen in the table, training the network using data gathered over 9 days dramatically increased the classification accuracy for multiple day test sets; in this case, the test sets consisted of data gathered on days 10 and 11. Therefore, training on days 1 through 9 allowed the networks to correctly verify target data from days 10 and 11 with a high degree of accuracy. Given the above results, the solution for verification over multiple days, which had not been addressed in previous literature, is a

Table 4.8. Classification Accuracies for Multiple day verification using data sets with 50% target images and 50% nontarget images. All training sets attained a 100% classification accuracy during training.

Data (day)	Hidden layer nodes	30 run average
day 1–4 vs 1–4	10	88.8%
day 1–4 vs 1–4	20	89.5%
day 1–4 vs 1–4	40	88.3%
day 1–4 vs 5	10	30.5%
day 1–4 vs 5	20	11.0%
day 1–4 vs 5	40	25.0%
day 1–9 vs 1–9	10	91.4%
day 1–9 vs 1–9	20	91.5%
day 1–9 vs 1–9	40	89.7%
day 1–9 vs 10	10	99.4%
day 1–9 vs 10	20	100.0%
day 1–9 vs 10	40	94.0%
day 1–9 vs 10–11	10	94.7%
day 1–9 vs 10–11	20	88.2%
day 1–9 vs 10–11	40	86.5%

backpropagation network with a single hidden layer and a training set with an equal number of target and nontarget data images collected over multiple days.

4.3.2.1 **False Acceptance Testing** This task was designed to test the false acceptance rate for nontargets. In other words, what percentage of the time will the network identify a nontarget as a target. The data for this test consisted of 90 images of nontargets, gathered over several days, and they were tested against the networks trained for day 1–9 data. Results of this testing are shown in Table 4.9. The table shows that for any number of hidden nodes used the false acceptance was no greater than 9.2 percent. As stated in the previous section, these results, and the verification results of Table 4.8, indicate that a backpropagation network with a single hidden layer can be used successfully for verification over multiple days.

Table 4.9. False acceptance testing of multiple day images of nontargets. Data was tested against networks previously trained for multiple day verification. The False Acceptance Rate indicates how often a nontarget was identified as a target, based on a 20 run average.

Data (day)	Hidden layer nodes	False Acceptance Rate
day 1–9 vs nontargets	10	6.9%
day 1–9 vs nontargets	20	9.6%
day 1–9 vs nontargets	40	8.2%

4.3.3 **Multiple Person Recognition** This portion of the testing used the backpropagation network once more to assess the ability of the network to classify multiple classes of face images. The parameters for the network remained as above. An immediate disadvantage of using the larger class networks was the training time. As a minimum, a 10 class problem with 40 hidden nodes required eight hours on the Sun SPARCstations to train and as the number of classes increased so did the training time to a maximum[1] time for these tests of 15 days for a 49 class problem with 250 hidden nodes. Results of the testing for this section are shown in Table 4.10.

For single day, multiple class problems the network could learn the training set and perform reasonable well on the test set accuracy, but when a second days data was added to the training and test set the accuracy was again a problem. No further testing of multiple day, multiple class could be performed to determine if additional prototypes would be useful because data for more than 2 days only existed for 3 individuals, all other subjects were 2 days of data only.

[1] Training to 200K iterations, which took 13 days and the training data only learned to 55.5%. Estimates of the time required to complete the training on Sun SPARCstations are 30 days.

Table 4.10. Classification Accuracies for Multiple Classes using raw image data (1024, 8-bit gray scale values) as inputs to a single hidden layer backpropagation network. All training sets attained 100% classification accuracy during training.

Classes	Hidden layer nodes	Iterations	Test Set Accuracy
10	10	10K	86.7%
10	20	10K	91.6%
10	30	10K	83.3%
31	31	30K	92.5%
31	62	50K	92.9%
31	93	50K	81.7%
49	50	100K	91.9%
49	100	100K	87.1%
49	150	200K	74.6%
49	250	200K	46.4%

4.4 Classification using raw data, Karhunen-Loève transform features, and identity network extracted features

For this task, multiple classes of data gathered over two days was tested in the neural networks described previously and in the AFIT end-to-end system (developed by Runyon) which uses the Karhunen-Loève Transform (KLT) as developed by Suarez (36). The purpose of this task was to compare the classification accuracy of a neural network using three different input features: raw image data, KLT extracted features, and identity network extracted features. The number of input features and hidden nodes used were determined from previous tests—these parameters gave the best results. The data sets for these tasks are as follows:

- Motion Segmented Data (MSD). This consisted of 230 images of 23 subjects (classes) captured over two days with the motion segmentation system described in (13).
- Hand Segmented Data (HSD). This data is composed of 600 images of 30 subjects captured over two days, one sitting per day, with the system described in chapter 3.

The identity network features were extracted from the motion segmented data and the hand segmented data and the reconstruction of images using these features is presented in Figure 4.6 for the motion segmentation and Figure 4.4 for the hand segmentation. Based on the visible reconstruction error, again refer to Figure 4.6, which was hypothesized to be important to some degree in face recognition, the features extracted from the motion segmented data should not do as well as the hand segmented data.

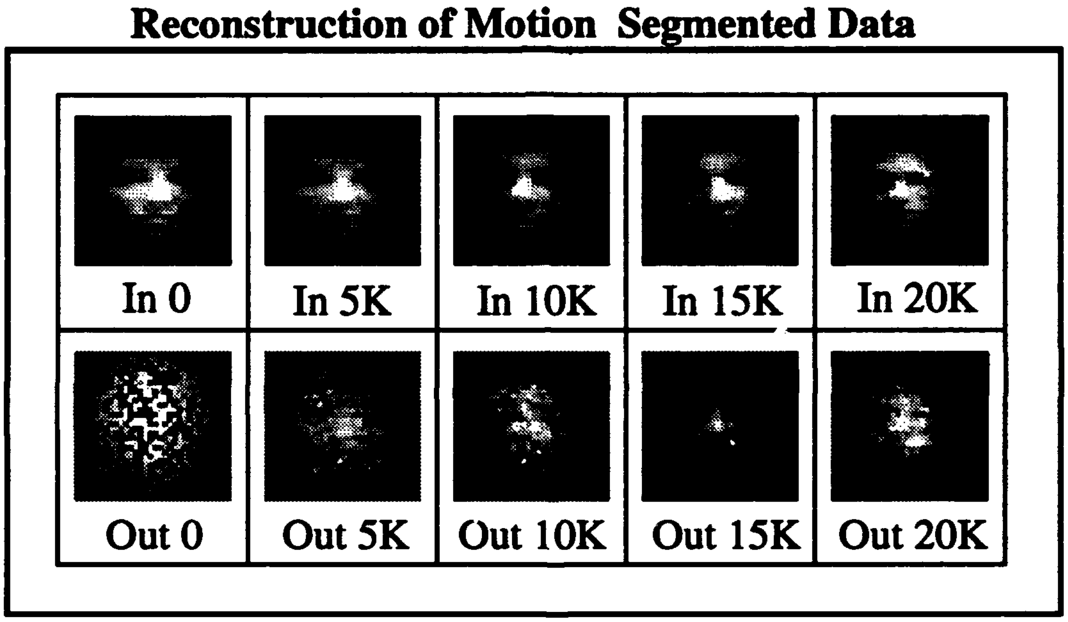

Figure 4.6. Identity network input images versus reconstructed output images using Motion segmented data. The images are shown at selected iterations of 0, 5000, 10000, 15000, and 20000. Although the identity network reduced the average MSE, the reconstruction of this data is not recognizable to a human observer.

The test results are shown in Table 4.11. Classification accuracy for hand segmented, single day, multiple classes is essentially equivalent for all types of inputs. However, using strict Karhunen-Loève transform features as inputs to a classification network yielded higher classification accuracies, by a few percentage points, than did the identity network extracted feaures as inputs to the classifier. This could be expected since the identity networks develop a distributed representation of the principal components and not necessarily the most significant principal components (11, 5) as does the strict KL transform. When using the motion segmented data, the Karhunen-Loève transform features were clearly better to use as inputs to a classification network. When combining data captured from two days, the results are again basically equivalent for the inputs used. Once more, however, the KLT extracted features gave a higher classification accuracy when using the motion segmented data. These tests clearly indicate that the KLT extracted features as inputs to a classification network provide the best results. The motion segmented data most likely gave lower results for the KLT features because the KLT is not shift or scale invariant and the motion segmented data is much more susceptable to these variations.

4.5 Code Development

As stated in chapter 3, all code was developed using ANSI Standard C. The algorithms developed as part of this research were tested using carefully developed test data files. Whenever possible, these files were run on the NeuralGraphics (37) system and code developed for this research to insure proper operation of the developed code.

4.6 Summary

This chapter presented the results of testing several different networks used for recognition or verification of face images. Networks tested consisted of a back propagation network using the raw data as inputs; a back propagation network using Karhunen-Loève Transform coefficients, computed from the raw data, as inputs; and a back propagation network using features extracted by an identity network as inputs. As discussed in Chapter 2, these networks had not been tested against data gathered

Table 4.11. Classification Accuracies for raw image data versus Karhunen-Loève transform extracted features versus identity network extracted features as inputs to a back propagation classification network. All training sets attained 100% classification accuracy during training. For this table, HSD is Hand Segmented Data and MSD is Motion Segmented Data

Classification Accuracies: Raw data vs KLT coefficients vs Identity network features as inputs				
	Network Configuration			
Data set	Inputs	Hidden Nodes	Outputs (classes)	Accuracy
HSD day 1	1024 (raw data)	60	30	93.3%
	20 (KLT coeffs)	40	30	97.0%
	40 (ID features)	40	30	93.3%
HSD day1+2	1024	60	30	93.3%
	20	40	30	95.0%
	40	40	30	93.8%
HSD day1vs2	1024	60	30	3.3%
	20	40	30	53.0%
	40	40	30	5.3%
MSD day1	1024	60	23	37.0%
	20	40	23	76.0%
	40	40	23	42.4%
MSD day1+2	1024	60	23	40.0%
	20	40	23	74.0%
	40	40	23	41.0%
MSD day1vs2	1024	60	23	10.4%
	20	40	23	34.0%
	40	40	23	13.9%

over multiple days. As such, this was a primary focus of the testing during this effort. The results show that all the networks perform poorly when training on data captured on a single day and testing on data gathered on a different day. However, the most important result for this effort is that verification can be accomplished over multiple days if the training set used contains data gathered over many days, in this case 9 days was sufficient, and if the training set is composed of 50 percent target images and 50 percent nontarget images. For this composition of training data, using a single hidden layer backpropagation network, and the raw image data as inputs, the verification accuracy over multiple days was 94 percent and the false acceptance for the same network was 6.5 percent. This indicates that face verification over multiple days can be performed with a neural network.

V. Conclusions

5.1 General

The purpose of this study was to investigate and implement a neural network for face verification and classification. The objective was to develop a neural network based feature extractor and/or classifier that can be used for authorized user verification in a realistic work environment. Specifically, three networks were developed and tested: a back propagation network using the raw data as inputs; a back propagation network using Karhunen-Loève Transform coefficients, computed from the raw data, as inputs; and a back propagation network using features extracted by an identity network as inputs. The following sections present the conclusions. First, the multiple day problem conclusion is presented, then the identity networks are discussed. This is followed by a discussion of the multiple class recognition problem; finally, the comparison of neural network extracted features to the strict Karhunen-Loève Transform features is presented.

5.2 Multiple Day Verification

The most significant conclusions for this research are based on the multiple day verification results. Multiple day verification is a significant problem that had not been addressed previously in the literature and the objective was to determine if a neural network could be used to solve the multiple day verification problem.

The solution to the multiple day verification problem, supported by test results of Chapter 4, is a single hidden layer backpropagation network that uses raw image data for inputs. When these networks were trained as two class networks accuracies of 100 percent were obtained for the multiple day data. It was also found that training set composition is important; the training data that provided the best results was a data set composed of 50 percent target individual, captured over 9 days, and 50 percent nontarget induviduals. Using less than 9 days worth of data proved to dramatically decrease the verification accuracy of the network. This is most likely due to the shift and scale sensitivity of these networks, also identified during testing. Gathering 9 days of images, ie. multiple prototypes, allowed the network to learn the shifted and scaled versions of the target which dramatically increased the verification accuracies. Again, these results are significant because the multiple day problem was not addressed in previous literature.

5.3 Identity Networks

The objective for this portion of the research was to develop and test an identity network for use as a feature extractor/input layer to a classifier. The data compression and feature extraction capabilities of these networks performed as expected. When using the identity network extracted features as inputs to a classifier, the features were acceptable for classification of data that had been gathered on a single day; 10 classes could be classified to a 97.5 percent accuracy. Increasing the class size had some effect on the classification accuracy using the identity network extracted features, however, the accuracy for 30 classes was 96.7 percent and for 49 classes it was 92.8 percent. These results are very respectable given the larger class sizes. However, the identity networks as feature extractors did not perform well when using data collected over multiple days. In this case, the maximum classification accuracy using multiple day data was 20.0 percent for a 10 class data set which indicates that the generalization capabilities of the identity networks over multiple days is very poor. As such, these networks would not be useful for feature extractors in a realistic environment.

5.4 Multiple Class Recognition

The multiple class recognition problem was addressed because many of the results in the literature deal with small numbers of classes and constrained data sets. The objective was to determine how well the networks performed when the data sets were increased in size, numbers of classes, and gathered over multiple days.

The classification accuracy using a single hidden layer backpropagation network with raw image data as inputs performed well on single day data. Results of classification for 310 images consisting of 31 classes, 10 face images per class, obtained a 92.9 percent classification accuracy on the single day data. Classification using two days of data consisting of 620 images of 31 classes, 10 images per class per day, also performed well for this network which obtained a classification accuracy of 93.3 percent. The drawback of this network, when using Sun workstations, is the amount of training time required. As the number of classes and hidden layer nodes increases the training time increases to a point of being impractical for real world applications to recognition. For example, a network using 800 images of 49 classes and a hidden layer of 250 nodes required over 30 days to train on the Sun Workstations. However, using 31 classes and 62 hidden nodes the network can be trainied in approximately 24 hours on Sun Workstations and this acceptable for realistic applications.

5.5 Neural Network Extracted Features versus KLT

The objective for this portion of the research was to compare the classification results of networks using raw data as inputs, identity network extracted features as inputs, or strict Karhunen-Loève Transform (KLT) extracted features as inputs. Data for these comparisons consisted of 300 images representing 30 classes for single day captured data and 600 images of 30 classes for two day captured data. The testing in this area showed that using the KLT extracted features as inputs to a classification network allowed for classification accuracies of 97% on single day captured data versus 93.3% for both the identity network extracted features and the raw data. When using the two day gathered data, the KLT features provided dramatic increases in classification accuracy over the single day data; 53% for the KLT extracted features versus 5.3% for the identity network extracted features and 3.3% for the raw data as inputs. The bottom line is that the neural networks using raw data or identity network extracted data as inputs to a classification network had accuracies as good as networks that used strict KLT coefficients as inputs.

Appendix A. Source Code

This appendix contains the source code for this research effort. The original backpropagation code and multilaye. perceptron code was written by Dr. (Capt) Dennis Ruck (29) and modified as it became necessary. The modifications included porting the code on Sun workstations, adding a symmetrical sigmoid update rule, implementing an identity network, and implementing a four layer network, etc. The following code is included in this appendix:

- **Makefile** The Makefile was setup to allow for variations in the executable files by setting or not setting various flags; the flags are self explanatory. Understanding the Makefile is key to understanding the code in this appendix.
- **dkmain.c** This is the main routine and contains many options depending on the particular flags defined.
- **backpropx.c** This is the backpropagation learning algorithm. This code was modified to include a symmetrical sigmoid updata rule as well as the standard sigmoid.
- **display.c** This code was developed to display images on the Sun workstation displays. It was specifically designed to view the identity network training while the training was in progress.
- **dkiox.c** This code contains the input and output routines for reading weight files, data files, etc.
- **ps.c, psx.c, utils.c** This code contains the utility routines necessary to gather the data in a meangful manner. The routines to compute the network, error statistics, etc. are contained in this code.
- **macros.h, globals.h, globals_.h** These are the '.h' files, the difference between globals.h and globals_.h is that globals.h is included in all files external to the main routine, dkmain.c.

- **makedata.c** This code is used to take the 32x32 pixel, 8 bit gray scale, ascii format image files and create the necessary data files for use in the main routine.

A.1 Makefile

```
# Makefile
# Created by Dennis Krepp June 1992
HOST=grimm
HOME=/cub5/dkrepp
NEXT_CFLAGS = -DNEXT -bsd
SUN_CFLAGS = -f68881 -DSUN
SUN4_CFLAGS = -bsd
ATHENA_CFLAGS = -ffpa -DSUN
DEC_CFLAGS = -DDEC
# The following flags apply primarily to the mfn.c program:
# -DMPX causes data to be partitioned according to training fraction input
# -DBACKPROP causes the Back Propagation training rule to be used
# (note: if training, this must be defined unless you
# code up another trainning rule )
# -DTRN causes the weights to be saved and the program to train a network.
# -DTRNCOTT causes the net to train the second layer of a Cottrell net
# -DNOTRN  use weight file weights for testing the network.
# -DLINEAROUT causes a network with linear outputs (Cybenko net)
# -DSIGMOID causes a network with sigmoidal output units
# -DSYM_SIGMOID causes network with symmetrical sigmoid outputs
# _DINP_SYM forces input layer to have a sym_sigmoid output
# -DSCALE_ETA causes eta to be scaled by the fan-in for each unit
# -DNOSCALE_ETA causes a fixed eta to be used for each weight
# -DMRUNS use to perform multiple runs on database
# -DNODE_OUT prints output node values for sampled iterations of input
# -DRANDOM causes random grabbing of data vectors for use in training
# -DIDNET causes the net to work as an autoassociator net
# (must define SYM_SIGMOID with IDNET)
# -DRESULTS causes outputs to be printed to files vs stdout
# -DMLP multilayer perceptron flag for output and setup files
# -DVIEW view the input and output images (IDNET must be defined also)
# -DOJA_ID makes an idnet function with linear outputs as suggested by Oja
# -DWTS forces a weight file to be saved after each output interval
SIGMOID_CFLAGS = -g -DNODEBUG -DBACKPROP -DRANDOM -DSIGMOID -DMPX \
 -DNOSCALE_ETA -DNOTRN -I.
IDNET_CFLAGS = -g -DNODEBUG -DBACKPROP -DRANDOM -DSYM_SIGMOID -DIDNET \
 -DNOSCALE_ETA -DTRN -DMPX -I.
CYBENKO_CFLAGS = -g -DNODEBUG -DBACKPROP -DRANDOM -DLINEAROUT -DMPX  \
 -DSCALE_ETA -DTRN -I.
COTTRELL_CFLAGS = -g -DNODEBUG -DBACKPROP -DRANDOM -DSYM_SIGMOID -DMPX  \
 -DNOSCALE_ETA -DTRNCOTT -I.
BIN = $(HOME)/bin/$(HOST)
#BIN=./
BACKPROP_OBJ = backpropx.c
MFN_CODE = $(BACKPROP_OBJ) dkmain.c ps.c dkiox.c utils.c psx.c nrutil.c
CC     = cc
SYS_LIB = -lm
mlp: Makefile globals_.h macros.h macros.h  nrutil.h malloc.h $(MFN_CODE)
$(CC) $(SIGMOID_CFLAGS) -DMLP -o mlp  $(MFN_CODE) $(SYS_LIB)
echo "Make mlp successfull! "

idnet: Makefile globals_.h macros.h macros.h nrutil.h malloc.h $(MFN_CODE)
$(CC) $(IDNET_CFLAGS) -o idnet  $(MFN_CODE) $(SYS_LIB)
echo "Make idnet successfull! "

cybenko: Makefile globals_.h macros.h macros.h nrutil.h malloc.h $(MFN_CODE)
$(CC) $(CYBENKO_CFLAGS) -o cybenko $(MFN_CODE) \
$(SYS_LIB)
echo "Make cybenko successfull! "

cottrell: Makefile globals_.h macros.h macros.h nrutil.h malloc.h $(MFN_CODE)
$(CC) $(COTTRELL_CFLAGS) -o cottrell $(MFN_CODE) \
$(SYS_LIB)
echo "Make cottrell successfull! "

stat_norm: Makefile stat_norm.c
$(CC) -g -I -o stat_norm stat_norm.c $(SYS_LIB)
echo "Make stat_norm successfull! "
NG2ruck: Makefile NG2ruck.c
$(CC) -g -I -o NG2ruck NG2ruck.c $(SYS_LIB)
echo "Make NG2ruck successfull! "

clean: ;rm -f *.o core
```

A.2 dkmain.c

```
/*****************************************************************

 dkmain.c: Multilayer Perceptron Program

 Written by:  Dennis W. Ruck (DS-90), AFIT/ENG
 Modified By: Dennis L. Krepp (GE-92D), AFIT/ENG
 Modifications:
  1. Modified to allow different learning rates (eta's)
     for each layer of weights.
  2. Modified to run as an Identity (autoassociator) net
  3. Modified to work as a Cottrell net
  4. Modified to use a symmetrical sigmoid if desired
  5. Modified to print output node values to screen if desired
  6. Modified to print the images to the screen if using the
  -DIDNET -DVIEW options
  7. Modified to work as a 4-layer classification net, which a
     slight twist to the Cottrell net in that the classification
   net is a two-layer
   backprop versus a one layer.
  8. Modified the multiple runs option of the code for SUNs

*****************************************************************/

#include <stdio.h>
#include <signal.h>
#include <ctype.h>

#ifndef NEXT
#include <malloc.h>
#else
 extern char * malloc();
#endif

#ifdef RESULTS
#define RUN_FILE run_file
#else
#define RUN_FILE stdout
#endif

#ifdef MLP
#define PREFIX "mlp"
#endif

#ifdef IDNET
#define PREFIX "idnet"
#define SIZE 32      /* this is the image size (ie. 32x32) */
#endif

#ifdef TRNCOTT
#define PREFIX "cottrell"
#endif

#ifdef TRNKREPP
#define PREFIX "dlk"
```

```
#endif

#ifdef LINEAROUT
#define PREFIX "cybenko"
#endif

#ifdef NODE_OUT
#define SHOW_OUTPUTS out_file
#endif

#include <macros.h>
#include <globals_h>

int     *ivector();
double  *dvector();
double   gaussian();
char    *get_token();
char    *make_file();
void     menu();
void     display_input();
void     display_output();
double   ones_normalize();

static  char     stop_name[] = "stop_file.dat";
FILE    *setup;
char    dat_temp[20];
char    wts_temp[20];
char    wts_filename[20];

main (argc, argv)
 int   argc;
 char *argv[];
{
 struct sigvec ivec;
 char command[80];


#ifdef MRUNS
#define SEEDNAME "seeds.dat"
 char       *run_root, *wts_root, *zoo_name;
 int        start_number, end_number;
 FILE        *seed_file;
 char        zoo_command[256];
 char       *run_name;
 char         temp[20];
 int        fnum;
#endif

 FILE *run_file;
 char      *wts_name, *dat_name;
 int       tng_vector, sample;
 double      err, acc, tst_err, tst_acc;
 FILE      *stop_file;
 FILE        *out_file;
 FILE      *plot_file;
 FILE      *idnet_file;
 char       inp_line[256];
 char      *SETUP_FILE, *NODE_FILE, *PLOT_FILE, *RESULTS_FILE;
 int       i, j, k, target, m;

/**** Create the file strings *********/

SETUP_FILE = make_file( "setup", PREFIX);
NODE_FILE  = make_file( PREFIX, "nodes");
PLOT_FILE  = make_file( PREFIX, "plot");
RESULTS_FILE = make_file( PREFIX, "results");


/***** Tell user what type this program is ****
This can be added to the output file later
if necessary
*************************************************/

#ifdef RESULTS
 RUN_FILE = fopen(RESULTS_FILE, "w");
```

```
#endif
#ifdef MPX
  fprintf(RUN_FILE,"mfn: MPX flag defined.\n");
#endif
#ifdef IDNET
  fprintf(RUN_FILE,"mfn: IDNET flag defined.\n");
#endif
#ifdef HOOX
  fprintf(RUN_FILE,"mfn: HOOX flag defined.\n");
#endif
#ifdef BACKPROP
  fprintf(RUN_FILE,"mfn: BACKPROP flag defined.\n");
#endif
#ifdef LINEAROUT
  fprintf(RUN_FILE,"mfn: LINEAROUT flag defined.\n");
#endif
#ifdef SIGMOID
  fprintf(RUN_FILE,"mfn: SIGMOID flag defined.\n");
#endif
#ifdef SYM_SIGMOID
  fprintf(RUN_FILE,"mfn: SYM_SIGMOID flag defined.\n");
#endif
#ifdef RANDOM
  fprintf(RUN_FILE,"mfn: RANDOM flag defined.\n");
#else
  fprintf(RUN_FILE,"mfn: RANDOM flag NOT defined.\n");
#endif
#ifdef MRUNS
  fprintf(RUN_FILE,"mfn: MRUNS flag defined.\n");
#endif
#ifdef SCALE_ETA
  fprintf(RUN_FILE,"mfn: SCALE_ETA flag defined.\n");
#endif
#ifdef TRNCOTT
  fprintf(RUN_FILE,"mfn: TRNCOTT flag defined.\n");
#endif
#ifdef TRNKREPP
  fprintf(RUN_FILE,"mfn: TRNKREPP flag defined.\n");
#endif


/***** Open the setup file *****/
if ((setup = fopen(SETUP_FILE, "r")) == NULL)
{
printf("I can't open the input file");
exit(-1);
}

/***** Read setup file for net_type *****/
fscanf(setup, "%d", &net_type);
 fskip_line(setup);

/***** Reading setup file *****/
  fscanf(setup, "%d", &initial_seed);
   fskip_line(setup);
  fscanf(setup, "%d", &part_seed);
   fskip_line(setup);
  fscanf(setup, "%d", &trn_seed );
   fskip_line(setup);

/***** Get filename to save weights to *****/
  fgets(wts_temp,20, setup);
   fskip.line(setup);
  i = 0;
  while(!isspace(wts_temp[i])) i++;
  wts_temp[i] = '\0';
  wts_name = wts_temp;


/***** Reading setup file *****/
  fscanf(setup, "%d", &max_iterations);
   fskip_line(setup);
  fscanf(setup,"%d %d %d %d",&num_inputs,&hide_one,&hide_two,&num_outputs);
   fskip_line(setup);
  ftr_list = ivector(0,num_inputs-1);
  mean = dvector(0,num_inputs-1);
  sd = dvector(0,num_inputs-1);
  input_mask = dvector(0,num_inputs-1);

/***** Get filename to read data from *****/
  fgets(dat_temp,20, setup);
   fskip_line(setup);
  i = 0;
  while(!isspace(dat_temp[i])) i++;
  dat_temp[i] = '\0';
  dat_name = dat_temp;


/***** Read setup file *****/
  fscanf(setup, "%d", &output_interval );
   fskip_line(setup);
  fscanf(setup, REAL_FMT, &eta_in );
   fskip_line(setup);
  fscanf(setup, REAL_FMT, &eta_out );
   fskip_line(setup);
  fscanf(setup, REAL_FMT, &eta_H1_H2 );
   fskip_line(setup);
  fscanf(setup, REAL_FMT, &alpha );
   fskip_line(setup);
  fscanf(setup, "%d", &batch_size );
   fskip_line(setup);
  fscanf(setup, REAL_FMT, &trn_frac );
   fskip_line(setup);
  norm_type = -1;
  fscanf(setup, "%d", &norm_type );
  fskip_line(setup);

#ifdef MRUNS
  fscanf(setup, "%d %d", &start_number, &end_number ); fskip_line(setup);
#endif



/** Close the Setup file **/
   fclose(setup);

#ifdef IDNET
  temp_outputs = num_outputs;
  num_outputs = num_inputs;
#endif

/***** INIT  THE NET **********/
  init_net();

/* fprintf(RUN_FILE, "\nUsing %d features in %s. ", num_inputs, dat_name);*/
  loopi(num_inputs) ftr_list[i] = i;



/*********************
  CREATE_FILE( stop_file, stop_name, "mfn" );
  fprintf( stop_file, "Delete this file to force a save of the weights.\n");
  fclose (stop_file);
*********************/

/***** READ THE DATA **************/
 read_dat( dat_name ); /** function found in dkiox.c **/
 fprintf(RUN_FILE, "\n%d vectors read", num_vectors);fflush(stdout);

#ifdef RESULTS
 fclose(RUN_FILE);
#endif

#ifdef MRUNS

if ((seed_file = fopen(SEEDNAME, "r")) == NULL)
{
printf("I can't open the seed file");
exit(-1);
}
```

```
  loopi(start_number) fscanf(seed_file, "%d %d %d", &initial_seed,
           &part_seed, &trn_seed );
   /** Multiple RUNS loop **/
  for(fnum = start_number; fnum≤end_number; fnum++) {

    iterations = 0;

    /** Create the file names **/
/*   sprintf(temp, "%s%d%s%s", PREFIX, fnum, ".", "RUN");
    run_name = temp;*/
    sprintf(temp, "%s%d%s%s", PREFIX, fnum, ".", "WTS");
    wts_name = temp;
/*   CREATE_FILE( run_file, run_name, "mmp2" );*/

    fscanf( seed_file, "%d %d %d", &initial_seed, &part_seed, &trn_seed );
#endif

/*******************************************
 Initialize/read the weights and thresholds
 -calls utils.c -> gaussian
 *******************************************/

#ifdef TRN
   srand48( initial_seed );
   loopi(num_states) xhat[i] = gaussian( 0.0, 1.0 );
   k_mpx_xfer();
#endif

#ifdef NOTRN
   read_wts(wts_name);
#endif

#ifdef TRNCOTT
   srand48( initial_seed );
   read_cottrell_wts(wts_name);
#endif

#ifdef TRNKREPP
   srand48( initial_seed );
   read_krepp_wts(wts_name);
#endif

   loopk(net_type)
   {
    loopi(Layer[k]→num_outputs)
    {
     loopj(Layer[k]→num_inputs)
   {
      Layer[k]→w_old[j][i] = Layer[k]→w[j][i];
      Layer[k]→dw[j][i] = 0.0;
   }
     Layer[k]→theta_old[i] = Layer[k]→theta[i];
     Layer[k]→dtheta[i] = 0.0;
    }
   }

 /** Partition the data into training and testing sets **/
#ifdef MPX
  partition( trn_frac, part_seed );
#endif

/********************
#ifdef IDNET
  idnet_partition ( part_seed);
#endif
********************/


#ifndef NOTRN
/***** Normalization of Data ********
  0: No Normalization.
  1: Gaussian Normalization.
  2: Normalize to range [-1,1]
 Select Normalization Criterion in
 setup file
*************************************/
 switch (norm_type) {
   case   0:  break;
   case   1:  gaussian_normalize ();
              break;
   case   2:  gaussian_normalize();
      max_value = ones_normalize();
      break;
   default:    printf("Invalid selection.\n");
              exit (-1);
              break;
 }
#endif

#ifdef NOTRN
 /** Normalize all data using previously generated mean/sd **/
 loopi(num_vectors) {
   loopj(num_inputs) {
    db_in[i][j] = (db_in[i][j]-mean[j])/sd[j];
   }
 }
#ifdef IDNET
 /* now divide all data by the max_value */
printf("\nmax_value = %f ", max_value);
 if (max_value ≠ 0.0)
 {
 loopi(num_vectors)
   loopj(num_inputs)
    db_in[i][j] = (db_in[i][j]/max_value);
 }
#endif
#endif

#ifdef RESULTS
 RUN_FILE = fopen(RESULTS_FILE, "a");
#endif

#ifdef IDNET
  loopij(num_vectors, temp_outputs)
    id_out[i][j] = db_out[i][j];
  loopij(num_vectors, num_outputs)
    db_out[i][j] = db_in[i][j];
fprintf(RUN_FILE, "\nNormalized IDNET output database initialized\n");
#endif /** end IDNET ifdef **/



  /** Print header for RUN file **/
  fprintf(RUN_FILE, "\n");
  fprintf(RUN_FILE, "Wt Selection seed (initial_seed): %u\n\
Db Partition seed (part_seed): %u\n\
Training Vector Selection seed(trn_seed): %u\n",
        initial_seed, part_seed, trn_seed );
  fprintf(RUN_FILE, "Weights file: %s\n", wts_name );
  fprintf(RUN_FILE, "Network Size: %d-%d-%d-%d\n",
        num_inputs, hide_one, hide_two, num_outputs );
  fprintf(RUN_FILE, "Source database: %s\n", dat_name );
  fprintf(RUN_FILE, "Training Rate (eta_in): %g\n\
Training Rate (eta_out): %g\n\
Training Rate (eta_H1_H2): %g\n\
Momentum (alpha): %g\n\
Batch Size: %d\n", eta_in, eta_out, eta_H1_H2, alpha, batch_size );
  fprintf(RUN_FILE, "Features Used: ");
  fprintf(RUN_FILE, "All Features in %s.\n", dat_name);
  fprintf(RUN_FILE, "\n");
  fprintf(RUN_FILE, "Fraction vectors assigned to training: %g\n", trn_frac );
  fprintf(RUN_FILE, "Normalization: %d\n", norm_type );
  fprintf(RUN_FILE, "\n");

  stats( &err, &acc, TRAIN );
  stats( &tst_err, &tst_acc, TEST );
  fprintf(RUN_FILE,"I: %6d, ERR: %6.4e, ACC: %6.4e, TSERR: %6.4e, TSACC: %6.4e\n", iterations,
     err, acc, tst_err, tst_acc );

 plot_file = fopen(PLOT_FILE, "a");
 fprintf(plot_file, "%6d %6.4e %6.4e %6.4e %6.4e \n", iterations,
     err, acc, tst_err, tst_acc );
 fclose(plot_file);

 #ifdef RESULTS
  fclose(RUN_FILE);
 #endif

 #ifdef NOTRN
 #ifdef IDNET
    idnet_file = fopen("mlp_idnet.dat", "w");
    fprintf(idnet_file, "%d\n%d\n", Layer[1]→num_outputs, temp_outputs);    loopj(num_vectors)
    {
    loopk(num_inputs) Layer[net_type−1]→X[k] = db_in[j][k];
    compute_output();
    fprintf(idnet_file, "%d ", j);
    loopi(Layer[1]→num_outputs)
       fprintf(idnet_file, "%lf ", Layer[1]→Y[i] );
    loopk(temp_outputs)
       fprintf(idnet_file, "%lf ", id_out[j][k] );
    fprintf(idnet_file, "\n");
    }
    fclose(idnet_file);
 #endif
 exit(−1);
 #endif


  /** training loop **/

  initstate( trn_seed, state, STATE_SIZE );

 #ifndef NOTRN
    save_wts( wts_name, dat_name );
 #endif

  /* sweeps loop */
  loopi( (max_iterations/num_trn)+1 ) {

    /** update loop **/
    loopj(num_trn) {

 #ifdef RANDOM
      target = trn_list[random()%num_trn];
 #else
      target = trn_list[iterations%num_trn];
 #endif
      loopm(num_vectors)
        if (vec_num[m] == target)
        break;
      current_vector = tng_vector = m;
      loopk(num_inputs) Layer[net_type−1]→X[k] = db_in[tng_vector][k];
      loopk(num_outputs) D_out[k] = db_out[tng_vector][k];

  /*** compute the outputs ****/

      compute_output();

 /** Display images if IDNET defined **/
 #ifdef IDNET
 if ((iterations % (int)(10.0*output_interval)) == 0)
 {
    display_input( tng_vector, max_value, SIZE);
    display_output( max_value, SIZE);
 }
 #endif
 /*************** NODE OUTPUT ***************/

 #ifdef NODE_OUT
 SHOW_OUTPUTS = fopen(NODE_FILE, "a");

/**** prints inputs and/or outputs up to a
  maximum of 10 nodes.  This can be changed
  to all nodes by changing the "if" statement
  below for the variable "sample"
 *****************************************/

if ((iterations % (int)(1.0*output_interval)) == 0){

 if (num_outputs > 10)
   sample = 10;
 else
   sample = num_outputs;

fprintf(SHOW_OUTPUTS, " Iteration %d:  Random vector %d:\n", iterations, target);

fprintf(SHOW_OUTPUTS, " Desired Outputs:  ");
loopk(sample) fprintf(SHOW_OUTPUTS, " %4.3f",db_out[tng_vector][k]);

#ifdef IDNET
fprintf(SHOW_OUTPUTS, "\n IDNET Outputs  :  ");
loopk(sample) fprintf(SHOW_OUTPUTS, " %4.3f", Layer[0]→Y[k]);
#else
fprintf(SHOW_OUTPUTS, "\n Actual Outputs :  ");
loopk(sample) fprintf(SHOW_OUTPUTS, " %4.3f", Layer[0]→Y[k]);
#endif
fprintf(SHOW_OUTPUTS, "  \n\n");
}

 fclose(SHOW_OUTPUTS);

#endif /******* END NODE_OUT ********/

#ifdef BACKPROP
    backprop(alpha);
#endif

    iterations++;

    if(iterations≥max_iterations) break;


#ifdef RESULTS
 RUN_FILE = fopen(RESULTS_FILE, "a");
#endif

   if((iterations % output_interval) == 0)
   {
     stats( &err, &acc, TRAIN );
     stats( &tst_err, &tst_acc, TEST );

       fprintf(RUN_FILE,"I: %6d, ERR: %6.4e, ACC: %6.4e, TSERR: %6.4e, TSACC: %6.4e\n", iterations, err,
acc, tst_err, tst_acc );


plot_file = fopen(PLOT_FILE, "a");
fprintf(plot_file, "%6d %6.4e %6.4e %6.4e %6.4e \n", iterations,
    err, acc, tst_err, tst_acc );
fclose(plot_file);

#ifdef RESULTS
 fclose(RUN_FILE);
#endif



#ifndef NOTRN
    save_wts( wts_name, dat_name );
#endif

/* Write weights to file after each epoch */
#ifdef WTS
sprintf(wts_filename, "%s%d", wts_name ,(int)(iterations/output_interval));
sprintf(command, "%s %s %s", "cp", wts_name, wts_filename);
system(command);
#endif


   }

#ifdef RESULTS
 fclose(RUN_FILE);
#endif




   } /** end update loop **/
   if(iterations≥max_iterations) break;
  } /** end sweeps loop **/

#ifndef NOTRN
 save_wts( wts_name, dat_name );
#endif

#ifdef RESULTS
 RUN_FILE = fopen(RESULTS_FILE, "a");
#endif

#ifdef IDNET /** Display last image **/
   display_input( tng_vector, max_value, SIZE);
   display_output( max_value, SIZE);
#endif

  stats( &err, &acc, TRAIN );
  stats( &tst_err, &tst_acc, TEST );

  fprintf(RUN_FILE,"I: %6d, ERR: %6.4e, ACC: %6.4e, TSERR: %6.4e, TSACC: %6.4e\n", iterations,
     err, acc, tst_err, tst_acc );

plot_file = fopen(PLOT_FILE, "a");
fprintf(plot_file, "%6d %6.4e %6.4e %6.4e %6.4e \n", iterations,
     err, acc, tst_err, tst_acc );
fclose(plot_file);

#ifdef RESULTS
 fclose(RUN_FILE);
#endif

/** write hidden node output to a file for mlp training **/
#ifdef IDNET
   idnet_file = fopen("mlp_idnet.dat", "w");
   fprintf(idnet_file, "%d\n%d\n", Layer[1]→num_outputs, temp_outputs);    loopj(num_vectors)
   {
   loopk(num_inputs) Layer[net_type−1]→X[k] = db_in[j][k];
   compute_output();
   fprintf(idnet_file, "%d ", j);
   loopi(Layer[1]→num_outputs)
      fprintf(idnet_file, "%lf ", Layer[1]→Y[i] );
   loopk(temp_outputs)
      fprintf(idnet_file, "%lf ", id_out[j][k] );
   fprintf(idnet_file, "\n");
   }
   fclose(idnet_file);
#endif



  switch (norm_type) {
   case   0:  break;
   case   1:  gaussian_unnormalize ();
              break;
  }

#ifdef RESULTS
 fclose(RUN_FILE);
#endif
```

```
#ifdef MRUNS
  }  /** end multiple runs loop **/
#endif

#ifdef VIEW
  system("rm *.red");
  system("rm temp.*");
  system("rm tempIN");
  system("rm tempOUT");
  system("rm *.rle");
#endif

#ifdef RESULTS
   fclose(RUN_FILE);
#endif

}
```

A.3 backpropx.c

```
/*****************************************************************

  backpropx.c: Routines supporting the backprop update rule for training
multilayer perceptrons. This version works with arbitrary numbers of layers
and uses the Layer[] array.

  Dennis W. Ruck, DS-90D
  AFIT/ENG

*****************************************************************/

#include <stdio.h>
#include <math.h>

#include <macros.h>
#include <globals.h>


void compute_dels();
void compute_delsx();
void backprop();
void backpropx();

void compute_dels()
{
/**********************************************************

  compute_dels: computes the deltas for a network. Uses net_type
  as global input to determine what layers to compute for.

**********************************************************/

switch (net_type) {
 case 1:
 case 2:
 case 3:    compute_delsx();
        break;
 default:  fprintf(stderr, "compute_dels: can't compute deltas for \
net_type = %d.\n", net_type);
        exit(-1);
        break;
}

} /** end compute_dels() **/

/*****************************************/
void compute_delsx()
{
 double   sum;
 int     i, j, k;

loopi(Layer[0]→num_outputs)

#ifdef LINEAROUT
   Layer[0]→del[i] = (D_out[i]−Layer[0]→Y[i])*Layer[0]→mask[i];
#endif

#ifdef OJA_ID
   Layer[0]→del[i] = (D_out[i]−Layer[0]→Y[i])*Layer[0]→mask[i];
#endif

#ifdef SYM_SIGMOID
   Layer[0]→del[i] = (0.5)*(1.0 − (Layer[0]→Y[i]*Layer[0]→Y[i]))*
           (D_out[i]−Layer[0]→Y[i])*Layer[0]→mask[i];
#endif

#ifdef SIGMOID
   Layer[0]→del[i] = Layer[0]→Y[i]*(1.0−Layer[0]→Y[i]) *
           (D_out[i]−Layer[0]→Y[i])*Layer[0]→mask[i];
#endif

for(k=1;k<net_type;k++)
{
 loopj(Layer[k]→num_outputs)
  {
   sum = 0.0;
   loopi(Layer[k−1]→num_outputs)
   sum += Layer[k−1]→del[i]*Layer[k−1]→w[j][i];
#ifdef SYM_SIGMOID
   Layer[k]→del[j] = (0.5)*(1.0 − (Layer[k]→Y[j]*Layer[k]→Y[j])) *
           sum*Layer[k]→mask[j];
#endif
#ifdef OJA_ID
   Layer[k]→del[j] = (0.5)*(1.0 − (Layer[k]→Y[j]*Layer[k]→Y[j])) *
           sum*Layer[k]→mask[j];
#endif
#ifdef SIGMOID
   Layer[k]→del[j] = Layer[k]→Y[j]*(1.0−Layer[k]→Y[j]) *
           sum*Layer[k]→mask[j];
#endif
#ifdef LINEAROUT
   Layer[k]→del[j] = Layer[k]→Y[j]*(1.0−Layer[k]→Y[j]) *
           sum*Layer[k]→mask[j];
#endif
  }
}

} /** end compute_delsx() **/

void backprop(alpha)
double alpha;
{
/**************************************************************

 backprop: Implement the backprop weight update rule.

INPUT: net_type

**************************************************************/

 switch(net_type){
  case   1:
  case   2:
  case   3: backpropx(alpha);
          break;
  default : fprintf(stderr,"backprop: Illegal net_type = %d\n",net_type);
        exit(−1);
 }
}

void backpropx(alpha)
double alpha;
{
```

```
/*************************************************************
  backpropx: Backprop update for a multilayer net.
 INPUTS: Layer[], D_out
 OUTPUTS: Layer[]
*************************************************************/

double    tW, tTheta;
double    leta, lalpha;
int       i, j, k, epoch;


lalpha = alpha;

batch_cnt++;

compute_dels();

/** Update all layers **/


#ifdef TRNCOTT
loopk((net_type - 1))
#endif
#ifdef TRNKREPP
loopk((net_type - 1))
#endif
#ifdef TRN
loopk(net_type)
#endif
{

#ifdef SCALE_ETA
  leta = Layer[k]→eta/(double)(Layer[k]→num_inputs);
#endif

#ifdef NOSCALE_ETA
  leta = Layer[k]→eta;
#endif

#ifdef IDNET
  epoch = (iterations/num_trn);
  if((epoch < 3) && (k == 0))
    leta = 0.30/(float)(1 + epoch);
  if((epoch > 25) && (k == 0))
    leta = (leta/10.0);

/*    if((iterations % num_trn) == 0)
      printf("\n epoch = %d, eta = %f\n",epoch,leta);
*/
#endif

/** Compute the change in the weights **/
 loopj(Layer[k]→num_outputs)
 {
  if(Layer[k]→mask[j]==OFF) continue;
  loopi(Layer[k]→num_inputs)
  {
   Layer[k]→dw[i][j] += Layer[k]→del[j]*Layer[k]→X[i];
  }
  Layer[k]→dtheta[j] += Layer[k]→del[j];
 }

 /** Update the weights if indicated **/
 if(batch_cnt==batch_size)
 {
  loopj(Layer[k]→num_outputs)
  {
   if(Layer[k]→mask[j]==OFF) continue;
   loopi(Layer[k]→num_inputs)
   {
    tW = Layer[k]→w[i][j];
       Layer[k]→w[i][j] += leta*Layer[k]→dw[i][j] +
                      lalpha*(Layer[k]→w[i][j]−Layer[k]→w_old[i][j]);
    Layer[k]→w_old[i][j] = tW;
    Layer[k]→dw[i][j] = 0.0;
     }
     tTheta = Layer[k]→theta[j];
     Layer[k]→theta[j] += leta*Layer[k]→dtheta[j] +
                      lalpha*(Layer[k]→theta[j]−Layer[k]→theta_old[j]);
     Layer[k]→theta_old[j] = tTheta;
     Layer[k]→dtheta[j] = 0.0;
    }
  } /** end weight update **/


} /** end layer update **/

if(batch_cnt==batch_size) batch_cnt = 0;

} /** end backpropx() **/
```

A.4 dkiox.c

```
/**************************************************

 dkiox.c: Utility Functions to support perceptron I/O

 Written by:  Dennis W. Ruck, DS-90D AFIT/ENG
 Modified by:  Dennis L. Krepp, GE-92D AFIT/ENG
 Modifications:
   1.  read_dat modified to save results to files
   2.  read_wts modified for a Cottrell net

*************************************************/

#include <stdio.h>
#include <strings.h>
#include <math.h>

#include "macros.h"
#include "globals.h"

#ifdef RESULTS
#define RUN_FILE stdout
#else
#define RUN_FILE stdout
#endif

double drand48();
double gaussian();
char * get_token();
double **dmatrix();
int *ivector();

/** Functions Exported **/
void read_dat();
void read_wts();
void read_cottrell_wts();
void save_wts();
void save_dat();
void make_mesh();
void save_xfm();


/****** VOID SAVE_WTS *************************************
 save_wts uses the following global input variables:

 num_inputs, hide_one, hide_two, num_outputs
 iterations, max_value
 initial_seed (used in setting weights)
 part_seed, trn_seed
 num_states
```

```
  net_type
  wts_type
  ftr_list
  norm_type
  trn_frac
  dominant_sensor
  num_flir, flir_list[]
  num_rng, rng_list[]

*******************************************************************/
void save_wts( wts_name, dat_name )
char   *wts_name, *dat_name[];
{

 FILE  *wts;
 char  *temp, *tkn;
 int    i = 0;


/* printf("kpiox.c-- save_wts \n");*/


  CREATE_FILE( wts, wts_name, "save_wts" );

  fprintf( wts, "WTS_TYPE: %d\n", wts_type );

  fprintf( wts, "%d -- Number inputs\n", num_inputs );
  fprintf( wts, "%d -- Number H1 nodes\n", hide_one );
  fprintf( wts, "%d -- Number H2 nodes\n", hide_two );
  fprintf( wts, "%d -- Number outputs\n", num_outputs );
  fprintf( wts, "%d -- Number iterations\n", iterations );
  fprintf( wts, "%d -- Initial wts seed\n", initial_seed );

 /*** Now save the weights ***/
 /*** Use net type ***/
 /***
   Type
    0  : Single perceptron
    1  : One layer
    2  : Two layer
    3  : Three layer
 ***/
 /*** Copy weights to the state array ***/
 switch(net_type) {

   case 0 :    fprintf(stderr,"save_wts: Illegal net_type = %d!\n", net_type);
         exit (-1);
         break;
   case 1 :
   case 2 :
   case 3 :    mpx_k_xfer();
         break;
   default:    fprintf(stderr,"save_wts: Illegal net_type = %d!\n", net_type);
         exit (-1);
         break;
  }

 loopi(num_states) fprintf( wts, "%e\n", xhat[i] );

 /** Save additional wts info for newer wts files **/
  if( wts_type == WTS_TYPE_1 )
  {
   fprintf( wts, "%d -- Partition seed\n", part_seed );
   fprintf( wts, "%d -- Training seed\n", trn_seed );
   fprintf( wts, "%s -- Source database\n", dat_name );
   fprintf( wts, "FEATURES: ");
   loopi(num_inputs) fprintf( wts, "%d\n", ftr_list[i] );
   fprintf( wts, "%d -- Normalization method\n", norm_type );
   fprintf( wts, "%f -- Normalization max_value\n", (float)max_value );
  fprintf( wts, "%e -- Fraction assigned to training\n", trn_frac );
   loopi(num_inputs)
    {
    fprintf( wts, "%f -- mean[%d]\n", (float) mean[i], i);
```

```
      fprintf( wts, "%f -- sd[%d]\n", (float) sd[i], i);
      }
   }

   fclose (wts);

} /* end save_wts */

/**************************************************/
void read_wts( wts_name )
char    *wts_name;
{
 FILE   *wts;
 char   *temp, *tkn, first;
 int     i = 0, tkn_len;
 double junk;

/*printf("dkiox.c-- read_wts \n");*/


  /*** Open the wts file ***/
  OPEN_FILE( wts, wts_name, "read_wts" );

  /** Check if the wts_type indicator is present **/
  if( (first=fgetc(wts)) ≠ 'W' ) {
    /** No wts_type indicator present **/
    /** Atempt to push it back onto the input stream **/
    if( ungetc(first, wts) == EOF ) {
      fprintf(stderr, "read_wts: can't return character to input stream.\n");
      exit (-1);
    }
    wts_type = WTS_TYPE_0;
  }
  else {
    /** wts_type indicator present **/
    tkn = get_token(wts); /** getting remainder of WTS_TYPE: label **/
    fscanf( wts, "%d", &wts_type );
    fskip_line(wts);
  }

  /*** Read the data ***/
  fscanf( wts, "%d", &num_inputs );
  fskip_line(wts);
  fscanf( wts, "%d", &hide_one );
  fskip_line(wts);
  fscanf( wts, "%d", &hide_two );
  fskip_line(wts);
  fscanf( wts, "%d", &num_outputs );
  fskip_line(wts);
  fscanf( wts, "%d", &iterations );
  fskip_line(wts);
  fscanf( wts, "%d", &initial_seed );
  fskip_line(wts);

  printf("num_inputs = %d \nhide_one = %d\nhide_two = %d\n\
num_outputs = %d\niterations = %d\ninitial_seed = %d\n\
wts_type = %d\n", num_inputs, hide_one, hide_two, num_outputs, iterations,
initial_seed, wts_type );

  /*** Now get the weights ***/
  /*** First determine net type ***/
  /***
    Type
     0  : Single perceptron
     1  : One layer
     2  : Two layer
     3  : Three layer
  ***/
  if ( (hide_one == 0) && (hide_two == 0) && (num_outputs == 1)) net_type = 0;
  else {
    if ( (hide_one == 0) && (hide_two == 0) ) net_type = 1;
  else {
    if ( (hide_two == 0) ) net_type = 2;
```

```
else net_type = 3;
   }}

printf("net_type = %d\n", net_type);
switch (net_type)
{
 case 0  :
 case 1  :     output_layer = &L1; break;
 case 2  :     output_layer = &L2;
            num_states = num_inputs*hide_one + hide_one +
                    hide_one*num_outputs+ num_outputs;
            break;
 case 3  :     output_layer = &L3;
            num_states = num_inputs*hide_one + hide_one +
                    hide_one*hide_two  + hide_two +
                    hide_two*num_outputs+ num_outputs;
            break;
}

/** Read in weights using the net_type **/

/*** First read into xhat array ***/
loopi(num_states) fscanf(wts, REAL_FMT, &xhat[i]);

/*** Copy weights to the layer structures ***/
switch(net_type) {

 case 0 :    break;
 case 1 :    break;
 case 2 :    k_mp2_xfer();
        break;
 case 3 :    k_mp3_xfer();
        break;
 default:    printf("read_wts: Illegal net_type = %d!\n", net_type);
        exit (-1);
        break;
 }

/** Read in additional wts info for newer wts files **/
if( wts_type == WTS_TYPE_1 ) {
  fscanf( wts, "%d", &part_seed ); fskip_line(wts);
  fscanf( wts, "%d", &trn_seed ); fskip_line(wts);
  fskip_line(wts);
  temp = get_token(wts); /** Skipping over "FEATURES:" **/
  loopi(num_inputs) fscanf( wts, "%d", &ftr_list[i] );
  fscanf( wts, "%d", &norm_type ); fskip_line(wts);
/*  fscanf( wts, REAL_FMT, &max_value ); fskip_line(wts);*/
  fscanf( wts, REAL_FMT, &junk ); fskip_line(wts);
  loopi(num_inputs)
   {
   fscanf( wts, REAL_FMT, &mean[i]);
   fskip_line(wts);
   fscanf( wts, REAL_FMT, &sd[i]);
   fskip_line(wts);
   }

 }

 fclose (wts);

} /* end read_wts */

/********* VOID READ_KREPP_WTS ********************/
void read_krepp_wts( wts_name )
char   *wts_name;
{
 FILE  *wts;
 char  *temp, *tkn, first;
 int   i = 0, tkn_len, junk;
 int   j;
 int   IDX = 0;


/*** Open the wts file ***/
OPEN_FILE( wts, wts_name, "read_wts" );
```

```
/** Check if the wts_type indicator is present **/
if( (first=fgetc(wts)) ≠ 'W' ) {
 /** No wts_type indicator present **/
 /** Atempt to push it back onto the input stream **/
 if( ungetc(first, wts) == EOF ) {
  fprintf(stderr, "read_wts: can't return character to input stream.\n");
  exit (−1);
 }
 wts_type = WTS_TYPE_0;
}
else {
 /** wts_type indicator present **/
 tkn = get_token(wts); /** getting remainder of WTS_TYPE: label **/
 fscanf( wts, "%d", &wts_type );
 fskip_line(wts);
}

/*** Read the data ***/
fscanf( wts, "%d", &num_inputs );
fskip_line(wts);
fscanf( wts, "%d", &hide_one );
fskip_line(wts);
fscanf( wts, "%d", &junk );
fskip_line(wts);
fscanf( wts, "%d", &junk );
fskip_line(wts);
fscanf( wts, "%d", &junk );
fskip_line(wts);
fscanf( wts, "%d", &junk );
fskip_line(wts);


/*** Now get the weights ***/
/*** Net type = Three layer Krepp ***/

output_layer = &L3;
num_states = num_inputs*hide_one + hide_one +
                    hide_one*hide_two  + hide_two +
                    hide_two*num_outputs+ num_outputs;


/** Read in weights using the net_type **/

/*** First read input layer wts into xhat array ***/
loopi((num_inputs*hide_one + hide_one))
  fscanf(wts, REAL_FMT, &xhat[i]);

/*** close the weight file ***/
fclose (wts);


/*** Copy weights to the layer structures. ****
** this code was copied from k_mp3_xfer and ***
** was specialized to read in the weights  ***
** for a Krepp net setup ****************/


/*** Get first layer ***/
loopi(hide_one)
{
 loopj(num_inputs) L1.w[j][i] = xhat[IDX++];
 L1.theta[i] = xhat[IDX++];
}

/*** Get second layer ***/
loopi(hide_two)
{
 loopj(hide_one) L2.w[j][i] = gaussian( 0.0, 1.0 );
 L2.theta[i] = gaussian( 0.0, 1.0 );
}

/*** Get third layer ***/
loopi(num_outputs)
```

```
 {
  loopj(hide_two) L3.w[j][i] = gaussian( 0.0, 1.0 );
  L3.theta[i] = gaussian( 0.0, 1.0 );
 }

 printf("\nKrepp output weights initialized \n");fflush(stdout);

} /* end read_krepp_wts */

/********* VOID READ_COTTRELL_WTS *******************/
void read_cottrell_wts( wts_name )
char    *wts_name;
{
 FILE   *wts;
 char   *temp, *tkn, first;
 int    i = 0, tkn_len, junk;
 int    j;
 int    IDX = 0;


  /*** Open the wts file ***/
  OPEN_FILE( wts, wts_name, "read_wts" );

  /** Check if the wts_type indicator is present **/
  if( (first=fgetc(wts)) != 'W') {
    /** No wts_type indicator present **/
    /** Atempt to push it back onto the input stream **/
    if( ungetc(first, wts) == EOF ) {
      fprintf(stderr, "read_wts: can't return character to input stream.\n");
      exit (-1);
    }
    wts_type = WTS_TYPE_0;
  }
  else {
    /** wts_type indicator present **/
    tkn = get_token(wts); /** getting remainder of WTS_TYPE: label **/
    fscanf( wts, "%d", &wts_type );
    fskip_line(wts);
  }

  /*** Read the data ***/
  fscanf( wts, "%d", &num_inputs );
  fskip_line(wts);
  fscanf( wts, "%d", &hide_one );
  fskip_line(wts);
  fscanf( wts, "%d", &hide_two );
  fskip_line(wts);
  fscanf( wts, "%d", &junk );
  fskip_line(wts);
  fscanf( wts, "%d", &junk );
  fskip_line(wts);
  fscanf( wts, "%d", &initial_seed );
  fskip_line(wts);


  /*** Now get the weights ***/
  /*** Net type = Two layer Cottrell ***/
  net_type = 2;

   output_layer = &L2;
   num_states = num_inputs*hide_one + hide_one +
                    hide_one*num_outputs+ num_outputs;

  /** Read in weights using the net_type **/

  /*** First read into xhat array ***/
  loopi(num_states) fscanf(wts, REAL_FMT, &xhat[i]);

  /*** close the weight file ***/
  fclose (wts);


 /*** Copy weights to the layer structures. ****
 ** this code was copied from k_mp2_xfer and ***
** was specialized to read in the weights  ***
** for a Cottrell net setup ****************/


/*** Get first layer ***/
loopi(hide_one)
{
 loopj(num_inputs) L1.w[j][i] = xhat[IDX++];
 L1.theta[i] = xhat[IDX++];
}

/*** Get second layer ***/
loopi(num_outputs)
{
 loopj(hide_one) L2.w[j][i] = gaussian( 0.0, 1.0 );
 L2.theta[i] = gaussian( 0.0, 1.0 );
}

printf("\nCottrell output weights initialized \n");fflush(stdout);

} /* end read_cottrell_wts */


/****** VOID READ_DAT ****************************************

 Read a database from file dat_name.

INPUTS:
  num_inputs: determines how many features to use.
  ftr_list[]: determines which features to use.
  num_outputs:  used if IDNET is defined.

OUTPUTS:
  num_vectors
  db_out[][]
  db_in[][]
  num_outputs
  vec_num[]
  vec_entry[]

**************************************************************/

void read_dat( dat_name )
char *dat_name;
{
FILE    *dat_file;
double   temp;
int      cnt = 0;
int      db_inputs, db_outputs;
int      i,j, inp_cnt, wcnt, est_vectors;
int scan_val;
char    *wctemp;



/** Open the file **/
 OPEN_FILE( dat_file, dat_name, "read_dat" );

#ifndef RESULTS
 fprintf(RUN_FILE, "\nReading %s: ", dat_name); fflush(stdout);
#endif

/** Read number of inputs and outputs **/
fscanf( dat_file, "%d", &db_inputs ); fskip_line(dat_file);
fscanf( dat_file, "%d", &db_outputs ); fskip_line(dat_file);

/** estimate the number of training vectors
   find the number of words and divide by (inputs+outputs+1) **/
wcnt = 0;
while(1){
 wctemp = get_token(dat_file);
 if(wctemp≠NULL) {
   wcnt++;
   free(wctemp);
  }
  else break;
 }
est_vectors = wcnt /(db_inputs+db_outputs+1);

#ifdef DEBUG
 fprintf(stderr,"Estimated number of vectors in %s is %d\n",dat_name,
   est_vectors);
#endif

/** Reset file for reading the vectors **/
rewind(dat_file); fskip_line(dat_file); fskip_line(dat_file);

/** Allocate db_in and db_out using the est_vectors+2 **/
db_in = dmatrix(0,est_vectors-1,0,num_inputs-1);
db_out = dmatrix(0,est_vectors-1,0,num_outputs-1);
vec_num = ivector(0,est_vectors-1);
vec_entry = ivector(0,est_vectors-1);
#ifdef IDNET
  id_out = dmatrix(0,est_vectors-1,0,db_outputs-1);
#endif

/** Read the training/test vectors **/
while( 1 )
{
 /** get next data pair **/
 if( (scan_val=fscanf( dat_file, "%d", &vec_num[cnt] )) == EOF)
 {
  num_vectors = cnt;
  break;
 }
 inp_cnt = 0;
 loopi(db_inputs)
 {
  if ( fscanf(dat_file, REAL_FMT, &temp) == EOF)
  {
   printf("read_dat: incomplete file - lacks full input vector.\n");
   exit (-1);
  }
  if( (inp_cnt<num_inputs) && (i == ftr_list[inp_cnt]) )
                              db_in[cnt][inp_cnt++] = temp;
 }
 loopi(db_outputs)
  if ( fscanf(dat_file, REAL_FMT, &db_out[cnt][i]) == EOF)
  {
   printf("read_dat: incomplete file - lacks full output vector.\n");
   exit (-1);
  }
 cnt++;

#ifndef RESULTS
 if(cnt%10==0) {fprintf(RUN_FILE,"*"); fflush(stdout);}
#endif

#ifdef DEBUG
 printf("scanf = %d, cnt = %d\n",scan_val, cnt);
#endif
} /** end while **/


if(est_vectors≠num_vectors) {
 fprintf(stderr,"read_dat: error in reading file: est_vectors = %d, \
num_vectors = %d\n",est_vectors, num_vectors);
 exit(-1);
}

fclose(dat_file);

} /* end read_dat */
```

```
void save_dat( dat_name )
char *dat_name;
{
/*********************************************************************
  save_dat: Routine to save a *.dat file. The purpose is to save the
  raw MSF db_in[][] and db_out[][] arrays on disk because the read_msfdat
  is a very slow process.

  INPUTS:

    db_in[][]
    db_out[][]
    vec_num[]
    num_inputs, num_outputs, num_vectors

  OUTPUTS:

    None.

*********************************************************************/

  int       j, k;
  FILE      *dat;

 printf("kpiox.c-- save_dat \n");

  CREATE_FILE( dat, dat_name, "save_dat" );

  fprintf( dat, "%d -- num_inputs\n%d -- num_outputs\n",num_inputs,
      num_outputs );

  loopk(num_vectors) {
    fprintf( dat, "%d ", vec_num[k]);
    loopj(num_inputs) fprintf( dat, "%e ", db_in[k][j] );
    loopj(num_outputs) fprintf( dat, "%e ", db_out[k][j] );
    fprintf( dat, "\n" );
  }

  fclose(dat);

} /* end save_dat */

void make_mesh( input, output)
double *input, *output;
{

/**************************************************************
- Procedure to generate test data for training and testing
- classification algorithms.

NOTE: It is assumed that the random number generators have been
set before calling this routine.

Use srand48(seedval) to initialize random number generator.

Output: Two Arrays of double. The first contains the input, and
the second contains the desired output.
**************************************************************/

#define LOW_OUT 0.10
#define HIGH_OUT 0.90

 int        class;
 double       radius;
 int        i;

printf("kpiox.c-- make_mesh \n");

 input[0] = drand48()*6.0 - 3.0;
 input[1] = drand48()*6.0 - 3.0;
```

```
  loopi(4) output[i] = LOW_OUT;

  /** - decide appropriate output for this vector **/
  radius = sqrt( input[0]*input[0] + input[1]*input[1] );
  if(radius > 2.5) {
    if(input[1]>0.0) /* class 3 */ output[2] = HIGH_OUT;
    else /* class 4 */ output[3] = HIGH_OUT;
  }
  else {
    if(radius<0.5) /* class 1 */ output[0] = HIGH_OUT;
    else {
      if(radius<1.5) {
        if(input[0]>0.0) /* class 2 */ output[1] = HIGH_OUT;
        else /* class 1 */ output[0] = HIGH_OUT;
      }
      else {
        if(input[0]>0.0) /* class 1 */ output[0] = HIGH_OUT;
        else /* class 2 */ output[1] = HIGH_OUT;
      }
    }
  }

}

void save_xfm( xfm_name )
char *xfm_name;
{
/****************************************************************

  INPUT: xfm_name
    net_type
    Layer[net_type-1]
    mean[], sd[]
    num_inputs

  OUTPUT: Data written to file

****************************************************************/

  int  i, j;
  FILE *xfm;


printf("kpiox.c-- save_xfm \n");


  CREATE_FILE( xfm, xfm_name, "save_xfm" );
  fprintf(xfm, "%3d -- num_inputs\n%3d -- num_outputs\n",
         Layer[net_type-1]→num_inputs,
         Layer[net_type-1]→num_outputs );

  /** Print the normalization data **/
  loopi(num_inputs) fprintf(xfm, "%E %E\n", mean[i], sd[i] );

  /** Print weights data **/
  loopi(Layer[net_type-1]→num_outputs) {
    loopj(num_inputs) fprintf(xfm, "%E ", Layer[net_type-1]→w[j][i] );
    fprintf(xfm, "%E\n", Layer[net_type-1]→theta[i] );
  }

  fclose(xfm);
} /* end save_xfm */
```

A.5 ps.c

```
/*************************************************

 ps.c:  Utility Functions to support perceptrons

 Dennis W. Ruck, DS-90D
 AFIT/ENG

*************************************************/

#include <stdio.h>
#include <math.h>

#include <macros.h>
#include <globals.h>


#ifdef RESULTS
#define RUN_FILE stdout
#else
#define RUN_FILE stdout
#endif

int    find_max();
void   sort();
int    *ivector();
double *dvector();
long labs();

void compute_output_from_H1();
void compute_output_from_H2();
void gaussian_normalize();
void gaussian_unnormalize();
double ones_normalize();
double sigmoid();
double symmetric_sigmoid();
void partition();
void stats();


/**********************************/
int k_mp2_xfer()
{
int i,j;
int IDX = 0;


/*printf("ps.c- k_mp2_xfer \n");*/


/*** Get first layer ***/
loopi(hide_one)
{
 loopj(num_inputs) L1.w[j][i] = xhat[IDX++];
 L1.theta[i] = xhat[IDX++];
}

/*** Get second layer ***/
loopi(num_outputs)
{
 loopj(hide_one) L2.w[j][i] = xhat[IDX++];
 L2.theta[i] = xhat[IDX++];
}

} /* end k_mp2_xfer */

/****************************/
int k_mp3_xfer()
{
int i,j;
int IDX = 0;

/*** Get first layer ***/
loopi(hide_one)
{
 loopj(num_inputs) L1.w[j][i] = xhat[IDX++];
 L1.theta[i] = xhat[IDX++];
}

/*** Get second layer ***/
loopi(hide_two)
{
 loopj(hide_one) L2.w[j][i] = xhat[IDX++];
 L2.theta[i] = xhat[IDX++];
}

/*** Get third layer ***/
loopi(num_outputs)
{
 loopj(hide_two) L3.w[j][i] = xhat[IDX++];
 L3.theta[i] = xhat[IDX++];
}

} /* end k_mp3_xfer */

/***********************************/
void stats(err, acc, set)
double *err;
double *acc;
int    set;
{

double sum = 0.0;
int err_cnt = 0;

int     *list, list_size, vec;
int i,j, target;


if( set == TRAIN ) {
  list = trn_list;
  list_size = num_trn;
  }
else {
  list = tst_list;
  list_size = num_tst;
  }
loopi(list_size)
{
  target = list[i];
  loopj(num_vectors)
   if (vec_num[j] == target)
     break;
  vec =  j;
  loopj(num_inputs) L1.X[j] = db_in[vec][j];

  compute_output();

  if ( find_max( &db_out[vec][0], num_outputs)
                              ≠ find_max( output_layer→Y, num_outputs ) )
   err_cnt++;
  loopj(num_outputs)
   sum += (db_out[vec][j] − output_layer→Y[j]) *
        (db_out[vec][j] − output_layer→Y[j]);

}

if(list_size == 0){
 *acc = 1.0;
 *err = 0.0;
  return;
}
*acc = ( (double) (list_size − err_cnt))/( (double) list_size);
*err = (0.5 * sum)/( (double) list_size);

} /* end stats */

/*********************************/
void compute_output_from_H1()
```

```
{

double sum;
int i,j;


printf("ps.c-- compute_output_from_H1 \n");


 switch (net_type) {
  case   2:
       break;
  case   3:
       break;
  default:    fprintf(stderr,
"compute_output_from_H1: can't compute for net_type = %d.\n",net_type);
       exit (-1);
       break;
 }

if(net_type == 3){
 /** Compute outputs of second hidden layer **/
 loopi(hide_two)
 {
  if(L2.mask[i]==OFF) {
  L3.X[i] = L2.Y[i] = 0.0;
  continue;
 }
 sum = 0.0;
  loopj(hide_one) sum += L2.X[j]*L2.w[j][i];
  sum += L2.theta[i];
  L3.X[i] = L2.Y[i] = sigmoid( sum );
 }
}

/** Compute outputs of final layer **/
loopi(num_outputs)
{
 if(output_layer→mask[i]==OFF) {
  output_layer→Y[i] = 0.0;
  continue;
 }
 sum = 0.0;
 loopj(hide_two) sum += output_layer→X[j]*output_layer→w[j][i];
 sum += output_layer→theta[i];
 output_layer→Y[i] = sigmoid( sum );
}

} /* end compute_output_from_H1 */

/*************************************/
void compute_output_from_H2()
{
double sum;
int i,j;



printf("ps.c-- compute_output_from_H2 \n");


 if(net_type ≠ 3) {
  fprintf(stderr, "compute_output_from_H2: can't compute for net_type = \
%d.\n", net_type );
  exit (-1);
 }

/** Compute outputs of final layer **/
loopi(num_outputs)
{
 if(L3.mask[i]==OFF) {
  L3.Y[i] = 0.0;
  continue;
 }
 sum = 0.0;
 loopj(hide_two) sum += L3.X[j]*L3.w[j][i];
 sum += L3.theta[i];
 L3.Y[i] = sigmoid( sum );
}

} /* end compute_output_from_H2 */

/***********************************/
double sigmoid ( a )
double a;
{
 double   max_exp;

 max_exp = 50.0;
 if ( a > max_exp ) return 1.0;
 if ( a < -max_exp ) return 0.0;
 return 1/(1 + exp(-a));
} /** end sigmoid **/

/***********************************/
double symmetric_sigmoid ( a )
double a;
{
 double   max_exp;

 max_exp = 50.0;
 if ( a > max_exp ) return 1.0;
 if ( a < -max_exp ) return -1.0;
 return ((2/(1 + exp(-a))) - 1);
} /** end symmetric_sigmoid **/

/****************************************
The following structure definition and
MATRIX definitions are for the
"idnet_partition" and "partition" functions
which follow
****************************************/
typedef struct src_data {
          int     vec_num;
            int     used;
        } src_data;

MATRIX_ALLOCATOR(src_data,matrix_src_data)
MATRIX_FREE(src_data,free_matrix_src_data)

/*************************************************************

  idnet_partition: separates the database into training and test sets.

*************************************************************/
void idnet_partition( part_seed )
long      part_seed;
{

 src_data  **src;
 int      i, j, class, cnt = 0, IDX;
 int      trn_cnt = 0;

 src = matrix_src_data(0,num_outputs-1,0,num_vectors-1);

/** Compute partitions **/
 num_trn = num_vectors;
 trn_list = ivector(0,num_trn-1);

/** Set up the class data in a format amenable to picking randomly
     from each class **/
 loopij(num_outputs,num_vectors) src[i][j].used = False;
 loopi(num_vectors) {
   class = 0;
   src[class][i].vec_num = vec_num[i];
  }
```

```
#ifdef DEBUG
   printf("Vectors assigned to class 0 are:\n");
   loopj(num_vectors){
    printf(" %d ", src[0][j]);
   }
   printf("\n");
#endif

/** Now actually assign the vectors to partitions **/

  initstate(part_seed, state, STATE_SIZE );

  loopj(num_vectors) {
    while (1) {
   if( src[0][(IDX = random() % num_vectors)].used == False ) {
    src[0][IDX].used = True;
    trn_list[trn_cnt] = src[0][IDX].vec_num;
    trn_cnt++;
    cnt++;
       break;
     }
    }
   }

  num_trn = trn_cnt;


#ifdef DEBUG
   printf("%d vector IDs assigned.\n",cnt );
#endif

#ifdef DEBUG
  fprintf(stderr, "The %d vectors in trn_list are: ", num_trn);
  loopi(num_trn) fprintf(stderr,"%d ", trn_list[i]);
  fprintf(stderr,"\n");
#endif

  free_matrix_src_data(src,0,num_outputs-1,0,num_vectors-1);

} /** end idnet_partition **/

/*****************************************************************

  partition: separates the database into training and test sets.

*****************************************************************/
void partition( trn_percent, part_seed )
double    trn_percent;
long      part_seed;
{

  src_data  **src;
  int       num_vecs[MAX_OUTPUTS];
  int       num_trnA[MAX_OUTPUTS];
  int       num_asgn[MAX_OUTPUTS];
  int       i, j, class, cnt = 0, IDX;
  int       trn_cnt = 0, tst_cnt = 0;

  src = matrix_src_data(0,num_outputs-1,0,num_vectors-1);

/** determine number of vectors of each class **/
  loopi(num_outputs) num_vecs[i] = 0;
  loopi(num_vectors) num_vecs[find_max(db_out[i],num_outputs)]++;

#ifdef DEBUG
  printf("The database contains %d total vectors.\n", num_vectors);
  loopi(num_outputs) printf("   %d in class %d\n", num_vecs[i], i );
#endif

/** Compute partitions **/
  loopi(num_outputs) num_trnA[i] = (int)( trn_percent*(double)num_vecs[i] );
  num_trn = 0;

#ifdef TRN
```

```
  loopi(num_outputs) num_trn += num_trnA[i];
  trn_list = ivector(0,num_trn-1);
#endif

  num_tst = num_vectors - num_trn;
  tst_list = ivector(0,num_tst-1);

#ifdef DEBUG
  printf("The number of vectors assigned to training by class are:\n");
   loopi(num_outputs) printf("%d training vectors in class %d.\n",
   num_trnA[i], i);
#endif

/** Set up the class data in a format amenable to picking randomly
      from each class ***/
  loopij(num_outputs,num_vectors) src[i][j].used = False;
  loopi(num_outputs) num_asgn[i] = 0;
  loopi(num_vectors) {
   class = find_max(db_out[i],num_outputs);
   src[class][num_asgn[class]++].vec_num = vec_num[i];
  }


#ifdef DEBUG
  loopi(num_outputs) {
   printf("Vectors assigned to class %d:", i);
   loopj(num_vecs[i])
    printf(" %d ", src[i][j]);
   printf("\n");
  }
#endif

#ifdef TRN
  /** Now actually assign the vectors to partitions **/
  initstate(part_seed, state, STATE_SIZE );
  loopi(num_outputs) {
   loopj(num_trnA[i]) {
    while (1) {
   if( src[i][(IDX = random() % num_vecs[i])].used == False ) {
    src[i][IDX].used = True;
    trn_list[trn_cnt] = src[i][IDX].vec_num;
    trn_cnt++;
    cnt++;
      break;
     }
    }
   }
  }

  num_trn = trn_cnt;
#endif



   loopi(num_outputs)
    loopj(num_vecs[i])
     if( src[i][j].used == False ) {
      tst_list[tst_cnt] = src[i][j].vec_num;
    tst_cnt++;
    cnt++;
     }

  num_tst = tst_cnt;

printf("\n%d test vectors assigned.", num_tst );

#ifdef DEBUG
  fprintf(stderr, "The %d vectors in trn_list are: ", num_trn);
  loopi(num_trn) fprintf(stderr,"%d ", trn_list[i]);
  fprintf(stderr,"\n");
  fprintf(stderr, "The %d vectors in tst_list are: ", num_tst);
  loopi(num_tst) fprintf(stderr,"%d ", tst_list[i]);
  fprintf(stderr,"\n");
```

```
#endif
 free_matrix_src_data(src,0,num_outputs-1,0,num_vectors-1);
}
void mp3_k_xfer()
{
/*****************************************************

 mp3_k_xfer: transfers the weights from the layered structure
to a vector structure for use with kalman training or saving data
to a file.

*****************************************************/

int      IDX = 0, i, j;

/**
 - The kalman algorithm assumes the weight matrix is of the form
 - W(dest,src);

 - The thresholds are put into the weight matrix as another column.
 - Thus the input vectors for each layer are augmented at the end
 - with an entry of unity.
 - The kalman vector is stored in row major form starting with the
 - first layer followed by the second layer and then the third layer.
**/

printf("ps.c-- mp3_k_xfer \n");

 loopi(hide_one) {
  loopj(num_inputs) xhat[IDX++] = L1.w[j][i];
  xhat[IDX++] = L1.theta[i];
 }
 loopi(hide_two) {
  loopj(hide_one) xhat[IDX++] = L2.w[j][i];
  xhat[IDX++] = L2.theta[i];
 }
 loopi(num_outputs) {
  loopj(hide_two) xhat[IDX++] = L3.w[j][i];
  xhat[IDX++] = L3.theta[i];
 }
}
void mp2_k_xfer()
{
/*****************************************************

 mp2_k_xfer: transfers the weights from the layered structure
to a vector structure for use with kalman training or saving data
to a file.

*****************************************************/

int      IDX = 0, i, j;

/**
 - The kalman algorithm assumes the weight matrix is of the form
 - W(dest,src);

 - The thresholds are put into the weight matrix as another column.
 - Thus the input vectors for each layer are augmented at the end
 - with an entry of unity.
 - The kalman vector is stored in row major form starting with the
 - first layer followed by the second layer and then the third layer.
**/

printf("ps.c-- mp2_k_xfer \n");
```

```
  loopi(hide_one) {
    loopj(num_inputs) xhat[IDX++] = L1.w[j][i];
    xhat[IDX++] = L1.theta[i];
  }
  loopi(num_outputs) {
    loopj(hide_one) xhat[IDX++] = L2.w[j][i];
    xhat[IDX++] = L2.theta[i];
  }
}

void gaussian_normalize()
{
/*****************************************************************

  gaussian_normalize(): Normalize data so that the training set has
a mean vector of zero and a standard deviation vector of all ones. Only
the features being used are normalized.

INPUTS:   num_trn
      trn_list[]
      vec_entry[]
      num_vectors
      num_inputs
      db_in[][]
        dominant_sensor

OUTPUTS:  db_in[][]
      mean[]
      sd[]

*****************************************************************/

 double      *sum, *sum_2;
 int         vec;
 int         i, j, k, lnum_inputs, target;

/*printf("ps.c- gaussian_normalize \n");*/

 sum = dvector(0,num_inputs-1);
 sum_2 = dvector(0,num_inputs-1);

 if((dominant_sensor==FLIR)||(dominant_sensor==RNG)) lnum_inputs = num_inputs-1;
  else lnum_inputs = num_inputs;

 loopi(lnum_inputs) sum[i] = sum_2[i] = 0.0;

/** Compute mean and sd of training data **/
 loopi(num_trn) {
  target = trn_list[i];
  loopj(num_vectors)
   if (target == vec_num[j])
   break;
  vec = j;
  loopj(lnum_inputs) {
   sum[j] += db_in[vec][j];
   sum_2[j] += db_in[vec][j]*db_in[vec][j];
  }
 }
 loopi(lnum_inputs) {
  mean[i] = sum[i] /(double) num_trn;
  sd[i] = sqrt( (sum_2[i] /(double) num_trn) - mean[i]*mean[i] );
  if (sd[i] == 0.00) sd[i] = 1.00;
 }

#ifdef DEBUG
 printf("Means: ");
 loopi(lnum_inputs) printf("%g ", mean[i]);
 printf("\n");
 printf("Std dev: ");
 loopi(lnum_inputs) printf("%g ", sd[i]);
 printf("\n");
#endif
```

```
/** Now apply to all data **/
loopi(num_vectors) {
  loopj(lnum_inputs) {
    db_in[i][j] = (db_in[i][j]-mean[j])/sd[j];
  }
}
free_dvector(sum,0,num_inputs-1);
free_dvector(sum_2,0,num_inputs-1);

} /** end gaussian_normalize **/

/*********************************************

  gaussian_unnormalize(): Unnormalize data.

INPUTS:  num_vectors
     num_inputs
     db_in[][]
        dominant_sensor
        mean[]
     sd[]

OUTPUTS:  db_in[][]

*********************************************/
void gaussian_unnormalize()
{

 int        i, j, lnum_inputs;

/*printf("ps.c- gaussian_unnormalize \n");*/

 if((dominant_sensor==FLIR)||(dominant_sensor==RNG)) lnum_inputs = num_inputs-1;
  else lnum_inputs = num_inputs;

 /** Now unnormalize all data **/
 loopij(num_vectors,lnum_inputs) db_in[i][j] = db_in[i][j]*sd[j] + mean[j];

} /** end gaussian_unnormalize **/

/************************************************
ones_normalize:  Normalizes data between 1 and -1.
             Must be called after the above
       gaussian_normalize routine is
       invoked.

INPUTS: db_in[][]
       num_inputs
    num_vectors

OUTPUT: max_value

************************************************/
double ones_normalize()
{

 int  i,j;
 double max_value1 = 0.0;
 double max_value2 = 0.0;
 double max;

 loopi(num_vectors)
 {
  loopj(num_inputs)
  {
   if ((max = fabs(db_in[i][j])) > max_value1)
     {
     max_value1 = max;
     }

   if ((max > max_value2) && (max ≠ max_value1))
     {
     max_value2 = max;
     }
  }
 }

fprintf(RUN_FILE,"\nones_normalize max_value1 is %9.6lf",max_value1);
fprintf(RUN_FILE,"\nones_normalize max_value2 is %9.6lf",max_value2);


/* now divide all data by the max_value */
 if (max_value1 ≠ 0.0)
 {
 loopi(num_vectors)
   loopj(num_inputs)
    db_in[i][j] = (db_in[i][j]/max_value1);
 }
 return max_value1;

} /* end ones_normalize */
```

A.6 psx.c

```
/*****************************************************************

  psx.c: Perceptron support package with routines that are dependent
  on the number of layers in the network.

  Dennis W. Ruck, AFIT/ENG
  DS-90D

*****************************************************************/

#include <stdio.h>
#include <math.h>

#include <macros.h>
#include <globals.h>

double sigmoid();
double symmetric_sigmoid();
void compute_output();
void compute_outputx();
void malloc_layer();
double **dmatrix();
double *dvector();
void display_input();
void display_output();
void display();


void init_net()
{
/*****************************************************************

  init_net(): Initializes data structures depending on the number
  of layers.

  Input: net_type, num_inputs, hide_one, hide_two, num_outputs,
      eta_in, eta_out, eta_H1_H2

  Output: Layer[], output_layer, num_states

*****************************************************************/

  int     i, j;

wts_type = WTS_TYPE_1;
loopi(num_inputs) input_mask[i] = ON;


/*printf("psx.c- init_net \n");*/
```

```
switch (net_type) {
 case 1:  hide_one = hide_two = 0;
      num_states = num_inputs*num_outputs + num_outputs;
         output_layer = &L1;
      Layer[0] = &L1;
      Layer[0]→num_outputs = num_outputs;
      Layer[0]→num_inputs = num_inputs;
      Layer[0]→eta = eta_out;
      malloc_layer( Layer[0],Layer[0]→num_inputs,Layer[0]→num_outputs );
      break;
 case 2:  hide_two = 0;
      num_states = num_inputs*hide_one + hide_one +
         hide_one*num_outputs + num_outputs;
      output_layer = &L2;
      Layer[0] = &L2;
      Layer[1] = &L1;
      Layer[0]→num_outputs = num_outputs;
      Layer[0]→num_inputs = hide_one;
      Layer[0]→eta = eta_out;
      Layer[1]→num_outputs = hide_one;
      Layer[1]→num_inputs = num_inputs;
      Layer[1]→eta = eta_in;
      malloc_layer( Layer[0],Layer[0]→num_inputs,Layer[0]→num_outputs );
      malloc_layer( Layer[1],Layer[1]→num_inputs,Layer[1]→num_outputs );
      break;
 case 3:
      num_states = num_inputs*hide_one + hide_one +
         hide_one*hide_two + hide_two +
         hide_two*num_outputs + num_outputs;
      output_layer = &L3;
      Layer[0] = &L3;
      Layer[1] = &L2;
      Layer[2] = &L1;
      Layer[0]→num_outputs = num_outputs;
      Layer[0]→num_inputs = hide_two;
      Layer[0]→eta = eta_out;
      Layer[1]→num_outputs = hide_two;
      Layer[1]→num_inputs = hide_one;
      Layer[1]→eta = eta_H1_H2;
      Layer[2]→num_outputs = hide_one;
      Layer[2]→num_inputs = num_inputs;
      Layer[2]→eta = eta_in;
      malloc_layer( Layer[0],Layer[0]→num_inputs,Layer[0]→num_outputs );
      malloc_layer( Layer[1],Layer[1]→num_inputs,Layer[1]→num_outputs );
      malloc_layer( Layer[2],Layer[2]→num_inputs,Layer[2]→num_outputs );
      break;
 default:  fprintf(stderr,"init_net: invalid net_type = %d\n", net_type);
      exit(−1);
      break;
}

xhat = dvector(0,num_states−1);
loopi(net_type) loopj(Layer[i]→num_outputs) Layer[i]→mask[j] = ON;

} /* end init_net */

/******************************************
k_mpx_xfer:  transfer weights from vector
 to a layered structure
******************************************/
void k_mpx_xfer()
{
int i,j, k;
int IDX = 0;


/*printf("psx.c- k_mpx_xfer \n");*/


rloopk(net_type) {

 /*** Get Next Layer ***/
 loopi(Layer[k]→num_outputs)
 {
    loopj(Layer[k]→num_inputs) Layer[k]→w[j][i] = xhat[IDX++];
    Layer[k]→theta[i] = xhat[IDX++];
  }
 }

} /* end k_mpx_xfer */

/*****************************************************

  mpx_k_xfer: transfers the weights from the layered structure
to a vector structure for use with kalman training or saving data
to a file.

*****************************************************/
void mpx_k_xfer()
{


int       IDX = 0, i, j, k;

/**
  - The kalman algorithm assumes the weight matrix is of the form
  - W(dest,src);

  - The thresholds are put into the weight matrix as another column.
  - Thus the input vectors for each layer are augmented at the end
  - with an entry of unity.
  - The kalman vector is stored in row major form starting with the
  - first layer followed by the second layer and then the third layer.
**/


/*printf("psx.c- mpx_k_xfer \n");*/


rloopk(net_type) {
 loopi(Layer[k]→num_outputs) {
   loopj(Layer[k]→num_inputs) xhat[IDX++] = Layer[k]→w[j][i];
   xhat[IDX++] = Layer[k]→theta[i];
 }
}
} /* end mpx_k_xfer */

/*******************************************/
void compute_output()
{

switch(net_type)
{
 case 1:
 case 2:
 case 3:  compute_outputx();
     break;
 default: printf("compute_output: can't perform calculation for net_type = \
%d\n", net_type);

}

} /* end compute_output */

/*******************************************/
void compute_outputx()
{

double sum, *mask;
int i,j, k;


 rloopk(net_type)
 {
  loopi(Layer[k]→num_outputs)
  {
   if(Layer[k]→mask[i]==OFF)
```

```
      {
      Layer[k]→Y[i] = 0.0;
      if(k≠0) Layer[k−1]→X[i] = 0.0;
      continue;
      }
     if(k==net_type−1) mask = input_mask;
     else mask = Layer[k+1]→mask;
     sum = 0.0;
     loopj(Layer[k]→num_inputs)
      sum += Layer[k]→X[j]*Layer[k]→w[j][i]*mask[j];
     sum += Layer[k]→theta[i];

#ifdef LINEAROUT
   if(k==0) Layer[k]→Y[i] =  sum;
   else Layer[k]→Y[i] = sigmoid( sum );
#endif

#ifdef SYM_SIGMOID
   Layer[k]→Y[i] = symmetric_sigmoid( sum );
#endif

#ifdef SIGMOID
   Layer[k]→Y[i] = sigmoid( sum );
#endif

#ifdef INP_SYM
   if(k==(net_type−1)) Layer[k]→Y[i] = symmetric_sigmoid( sum );
#endif

   if(k≠0) Layer[k−1]→X[i] = Layer[k]→Y[i];
  }
 }

} /* end compute_outputx */

/**********************************************/
void malloc_layer( L, inputs, outputs )
struct layer *L;
int inputs, outputs;
{

/* printf("psx.c- malloc_layer \n");*/


 L→w = dmatrix(0,inputs−1,0,outputs−1);
 L→dw = dmatrix(0,inputs−1,0,outputs−1);
 L→w_old = dmatrix(0,inputs−1,0,outputs−1);
 L→theta = dvector(0,outputs−1);
 L→dtheta = dvector(0,outputs−1);
 L→theta_old = dvector(0,outputs−1);
 L→beta = dvector(0,outputs−1);
 L→gamma = dvector(0,outputs−1);
 L→del = dvector(0,outputs−1);
 L→mask = dvector(0,outputs−1);
 L→X = dvector(0,inputs−1);
 L→Y = dvector(0,outputs−1);

}

/**********************************************/
void display_input(number, max, size)
 int number, size;
 double max;
{
 FILE *image;
 double temp;
 int j;
 static int num_in = 0;
 char infile[20];
 char command[80];


 num_in++;
 sprintf(infile, "%s%d%s", "IN", num_in, ".rec");
 image = fopen(infile, "w");
 loopj(num_inputs)
  {
   temp = ((db_in[number][j]*max)*sd[j]) + mean[j];
   fprintf(image, "%10.6f ", temp);
  }
 fclose(image);
#ifdef VIEW
 sprintf(command, "%s%s%s", "cp ", infile, " tempIN");
 system(command);
 display(size, "tempIN");
#endif
}


/***********************************************/
void display_output(max, size)
 double max;
 int size;
{
 FILE *image;
 double temp;
 int i;
 static int num_out = 0;
 char outfile[20];
 char command[80];
 double sqrt();


 num_out++;
 sprintf(outfile, "%s%d%s", "OUT", num_out, ".rec");

 image = fopen(outfile, "w");
 loopi(num_outputs)
  {
   temp = ((Layer[0]→Y[i]*max)*sd[i]) + mean[i];
   fprintf(image, "%10.6f ", temp);
  }
 fclose(image);
#ifdef VIEW
 sprintf(command, "%s%s%s", "cp ", outfile, " tempOUT");
 system(command);
 display(size, "tempOUT");
#endif
}


/**************************************
*  display.c
*  converts a .gra file to rle and
*  displays it in openwindows using
*  xli.
***************************************/

void display( X, filename)
 int X;
 char filename[];
{

char      command[80];

  sprintf(command, "%s%s%s", "cp ", filename," temp.rec");
  system(command);
  system("float_gray temp.rec temp.red");
  sprintf(command, "%s%d %d", "graytorle -o temp.rle ", X, X);
  strcat(command, " temp.red");
  system(command);
  sprintf(command,"%s%s%s","mv temp.rle ", filename,".rle");
  system(command);
  sprintf(command,"%s%s%s","xli -quiet -zoom 300 ", filename, ".rle &");
  system(command);
}
```

A.7 utils.c

```
/**************************************************

  Utility Functions

**************************************************/

#include <stdio.h>

#ifndef NEXT
#include <malloc.h>
#else
 extern char * malloc();
#endif

#include <math.h>
#include <string.h>

#include <macros.h>
#include <globals.h>

/** External System Calls **/
extern char * getenv();
double drand48 ();

/** Internal Functions **/
int   find_max();
Boolean find();
void  system_check();
void sort();
char * parse_fname();
char * get_token ();
double gaussian();
double distance2();
char * make_name();

/*********************************************/
int find_max( data, len )
double *data;
int   len;
{

double max_val = *data;
int   max_idx = 0;
int i;

loopi(len) if ( *(data+i) > max_val )
      {
       max_val = *(data+i);
       max_idx = i;
         }
return max_idx;

} /* end find_max */

/*********************************************/
void system_check()
/** Checks the system for reliable operations **/
{
 static char    sc_name[80] = "system_check1A992";
 FILE   *sc;
 double  x_out = 123.321;
 double  x_in;


printf("utils.c-- system_check \n");


 /* I/O Check */
 if( (sc = fopen(sc_name, "w")) == NULL)
 {
  fprintf( stderr, "system_check: can't open %s for writing.\n\
FATAL Error.\n", sc_name );
```

```
  exit (−1);
 }

 fprintf( sc, REAL_FMT, x_out );
 fclose (sc);

 if( (sc = fopen(sc_name, "r")) == NULL)
 {
  fprintf( stderr, "system_check: can't open %s for reading.\n\
FATAL Error.\n", sc_name );
  exit (−1);
 }

 fscanf( sc, REAL_FMT, &x_in );
 if( x_out ≠ x_in )
 {
  fprintf( stderr, "system_check: Floating Point I/O Error.\n\
FATAL Error.\n" );
  unlink( sc_name );
  exit (−1);
 }

 unlink ( sc_name );
 printf( "system_check: OK.\n" );

} /* end system_check */

/** Procedure to sort an input array **/

void Qpartition ( data, rank, split, lower, upper, new_l, new_u )
double     *data;
int        *rank;
register int split;
int        lower;
int        upper;
int        *new_l;
int        *new_u;
{

 register int i = lower, j = upper;
 register int temp;


printf("utils.c-- Qpartition \n");


 do {
    while ( data[rank[i]] < data[split] ) i++;
    while ( data[split] < data[rank[j]] ) j−−;
    if ( j ≥ i ) {
   temp = rank[i];
   rank[i] = rank[j];
   rank[j] = temp;
   i++; j−−;
    }
   } while (i<j);
   *new_l = j;
   *new_u = i;
}

/** Recursive quicksort **/
void quicksort( data, rank, upper, lower )
double    *data;
int       *rank;
int       upper, lower;
{
 int      split;
 int      new_up, new_low;
 int      temp;
 int      i;


printf("utils.c-- quicksort \n");
```

```
  if ( upper−lower > 1 ) {
    split = rank[(upper+lower)/2];
    Qpartition( data, rank, split, lower, upper, &new_low, &new_up );
    quicksort( data, rank, upper, new_up );
    quicksort( data, rank, new_low, lower );
  } else
   if (upper − lower == 1)
     if ( data[rank[upper]] < data[rank[lower]] ) {
    temp = rank[upper];
    rank[upper] = rank[lower];
    rank[lower] = temp;
     }
}

void sort( data, num_elements, rank )
double    *data;
int       num_elements;
int       *rank;
{
/** The output is the rank of integers, rank, which gives the sorted
order of the the rank of doubles, data. That is, the smallest entry in
the array data is index rank[0].                                  */

  int     i;


printf("utils.c-- sort \n");


  loopi(num_elements) rank[i] = i;

  quicksort( data, rank, num_elements−1, 0 );

}

/***********************************************************/
char * parse_fname( fname )
char *fname;
/***********************************************************/
{
 char * out_name;
 char   var_name[256];
 char   out_temp[256];
 char * var_path;
 int    fn_len, vn_len, vp_len, out_len;
 int    i;


printf("utils.c-- parse_fname \n");


 fn_len = strlen(fname);

 /** See if it starts with a dollar sign */
 if( fname[0] == '$' )
 {
  /** expand environment variable */
  i = 1;
  while( fname[i] ≠ '/' )
  {
   var_name[i−1] = fname[i];
   i++;
  }
  var_name[i−1] = '\0';
  if( (var_path = getenv(var_name)) == NULL)
  {
   printf("parse_fname: environment  ..iable $%s. Not defined",
        var_name);
   exit (−1);
  }
  /** Now combine into the full path */
  vn_len = strlen(var_name);
  vp_len = strlen(var_path);
```

```
   loopi(vp_len) out_temp[i] = var_path[i];

   for(i=0;i<fn_len−vn_len−1;i++)
   out_temp[i+vp_len] = fname[i+vn_len+1];
   out_temp[i+vp_len] = '\0';
  }
  else
  {
   /** copy input path to output path **/
   loopi( fn_len+1 ) out_temp[i] = fname[i];
  }

  /** Now allocate memory for out_name **/
  if( (out_name = malloc( (out_len = strlen(out_temp))+1 )) == NULL)
  {
   printf("parse_fname: out of memory.\n");
   exit (−1);
  }
  loopi(out_len+1) out_name[i] = out_temp[i];

  return out_name;

} /* end parse_fname */

/*************************************************************/
char * get_token( str )
FILE *str;
/*************************************************************/
/*
  Returns the next string of characters in stream, STR, which is separated
  with white space.

**************************************************************/
{

  char  temp[1024];
  char * tk_ptr;

  int   i;

  /** Find first character of token **/
  while(1)
  {
   if( (temp[0] = fgetc(str)) == EOF) return NULL;
   if( (temp[0] ≠ ' ') && (temp[0] ≠ '\n') ) break;
  }

  i=1;
  while(1)
  {
   temp[i] = fgetc(str);
   switch (temp[i])
   {
    case ' ':
    case '\n':    temp[i] = '\0';
         break;
    case EOF :   temp[i] = '\0';
         break;
    default :    break;
    }
   if( temp[i] == '\0' ) break;
   i++;
  }
 if( (tk_ptr = malloc( strlen(temp)+1 )) == NULL){
   fprintf( stderr, "get_token: out of memory.\n" );
   exit (−1); }
 loopi( strlen(temp)+1 ) tk_ptr[i] = temp[i];
 return tk_ptr;

} /* end get_token */
```

```
double gaussian( mean, var )
double  mean, var;
{
/**********************************************************************

 gaussian: returns a gaussian randon variable sample with specified
mean and variance. The central limit theorem is invoked to
generate the sample.

**********************************************************************/


int      num_rvs = 20, i;
double   sum = 0.0, ave, norm, Z, Y;


/****
 - Obtain a sum of random variables that are uniform between
 - 0 and 1.
****/
 loopi(num_rvs) sum += drand48();

 ave = sum / (double)(num_rvs);

/**
 - AVE is a rv with mean = 0.5 and variance = 1/(12*num_rvs);
 - now normalize AVE
**/
 Z = (ave-0.5)/sqrt(1.0/(12.0*(double)(num_rvs)));

/**
 - Now unnormalize to desired mean and variance
**/
 Y = mean + sqrt(var)*Z;

 return Y;
}

/**********************************************************/
char *make_name( num, root, ext )
 int   num;
 char  *root, *ext;
{
/**********************************************************

 make_name: Function to create a file name given the root,
 a number, and the extension. The file name is of the form:

 root number "." ext

**********************************************************/

 char    *fname, num_image[80];


printf("utils.c-- make_name \n");


 sprintf( num_image, "%d", num );
 fname = malloc( strlen(root)+strlen(num_image)+strlen(ext)+2 );
 strcpy( fname, root );
 strcat( fname, num_image );
 strcat( fname, "." );
 strcat( fname, ext );

 return fname;
} /* end make_name */


/**********************************************************/
char * make_file( root, ext )
 char  *root, *ext;
{
/**********************************************************

 make_file: Function to create a file name given the root
 and the extension. The file name is of the form:

 root "." ext

***********************************************************/

 char    *fname, num_image[80];

 fname = malloc( strlen(root)+strlen(ext)+2 );
 strcpy( fname, root );
 strcat( fname, "." );
 strcat( fname, ext );

 return fname;
} /* end make_file */



/*********************************************************/
Boolean find(run_file, var_name)
FILE *run_file;
char *var_name;
{
 int    cnt, var_length;
 char   str[80];


printf("utils.c-- find \n");


 var_length = strlen(var_name);

 while(1) {
  if((str[0]=fgetc(run_file))==EOF) break;
  if(str[0]==var_name[0]) {
    cnt = 1;
    while(1) {
  if( ((str[cnt]=fgetc(run_file))==EOF) ||
      (cnt≥var_length) ) break;
      cnt++;
    }
    str[var_length] = '\0';
    if(strcmp(str,var_name)==0) return True;
  }
 }

 return False;

} /** end find **/

double distance2( x1, x2, len )
double x1[], x2[];
int    len;
{
/*****************************************************************

  distance2: compute Euclidean distance between to vectors

*****************************************************************/

 double sum;
 int    i;


printf("utils.c-- distance2 \n");


 sum = 0.0;
 loopi(len) sum += (x1[i]-x2[i])*(x1[i]-x2[i]);
 return sqrt(sum);

} /** end distance2 **/

#ifdef NEXT
```

```
double drand48()
{
 long  lval;

 lval = random()%2147483648;
 return (double)lval/2147483648.0;
}

void srand48(seed)
long seed;
{
 random(seed);
}
#endif
```

A.8 display.c

```
/*************************************
*   display.c
*   converts a .gra file to rle and
*   displays it in openwindows using
*   xli.
*************************************/

void display(dimension, filename)
 int dimension;
 char filename[];
{

char      command[80];

  sprintf(command, "%s%s%s", "cp ", filename,".gra temp.rec");
  system(command);
  system("float_gray temp.rec temp.red");
  switch(dimension){
    case 128:
     system("graytorle -o temp.rle 128 128 temp.red");
   break;
    case 64:
     system("graytorle -o temp.rle 64 64 temp.red");
   break;
    case 32:
     system("graytorle -o temp.rle 32 32 temp.red");
   break;
    default:
   printf("I don't know what size the gra image is.");
  }
  system("rleflip -v -o hold.rle temp.rle");
  sprintf(command,"%s%s%s","mv hold.rle ", filename,".rle");
  system(command);

  sprintf(command,"%s%s%s","xli -quiet -zoom 300 -smooth -smooth ", filename, ".rle&");

  system(command);
  system("rm *.red");
  system("rm temp.*");
  system("rm *.rle");
}
```

A.9 globals_h

```
/**** Global Variables ***/
  int num_inputs, hide_one, hide_two, num_outputs, temp_outputs;
  double eta_in, eta_out, eta_H1_H2;
  int net_type;
  int wts_type = WTS_TYPE_1;
  int num_layers;
  int iterations = 0;
  int initial_seed, part_seed, trn_seed;
  int num_states;
  int num_vectors = 0;
  int *ftr_list;
  int *vec_num;
  int *vec_entry;
  int num_trn = 0, *trn_list;
  int num_tst = 0, *tst_list;
  int norm_type;
  int dominant_sensor = 0;
  int num_flir, *flir_list;
  int num_rng, *rng_list;
  int current_vector;
  int max_iterations, output_interval;
  int batch_size = 1, batch_cnt = 0;

  char state[STATE_SIZE];

  double trn_frac;
  double alpha;
  double max_value;
 struct layer {
      int       num_inputs;
      int       num_outputs;
      double    eta;
      double    **w;
      double    **dw;
      double    **w_old;
      double    *theta;
      double    *dtheta;
      double    *theta_old;
      double    *del;
      double    *beta;
      double    *gamma;
      double    *mask;
      double    *X;
      double    *Y;
          } L1, L2, L3;
 double *input_mask;
 double *xhat;
 double D_out[MAX_OUTPUTS];
 struct layer *output_layer, *Layer[3];
 double **db_in;
 double **db_out;
 double **id_out;
 double *mean, *sd;

/** Global Data for Kalman Training **/
double   **R, d[MAX_OUTPUTS], z[MAX_OUTPUTS];
```

A.10 macros.h

```
/*****************************************************

Convenient Macros for Perceptron Package

*****************************************************/

/*** MACROS ***/

#define REAL float

#ifndef GCC
#define INT_MAX (2147483647)
```

```
#else /* GCC */
#include <limits.h>
#endif

#ifdef VMS
#define unlink delete
#endif

#ifdef LEO
#define REAL_FMT "%g"
#else
#define REAL_FMT "%lg"
#endif
#ifdef NEXT
#undef REAL_FMT
#define REAL_FMT "%lf"
#endif
#ifdef VMS
#undef REAL_FMT
#define REAL_FMT "%lf"
#endif

#define Boolean int
#define False 0
#define True  1

/** Dominant Sensor Definitions **/
#define SINGLE 0
#define FLIR 1
#define RNG  2

/** Mask Definitions **/
#define OFF 0.0
#define ON  1.0

char     junk_response[256];

#define fskip_line(A) fgets(junk_response, 256, A)
#define skip_line gets(junk_response)
#define rloopi(A) for(i=(A)-1;i≥0;i--)
#define rloopj(A) for(j=(A)-1;j≥0;j--)
#define rloopk(A) for(k=(A)-1;k≥0;k--)
#define rloopl(A) for(l=(A)-1;l≥0;l--)
#define rloopm(A) for(m=(A)-1;m≥0;m--)
#define rloopn(A) for(n=(A)-1;n≥0;n--)
#define rloopp(A) for(p=(A)-1;p≥0;p--)
#define rloopij(A,B) for(i=(A)-1;i≥0;i--) for(j=(B)-1;j≥0;j--)
#define loopi(A) for(i=0;i<A;i++)
#define loopj(A) for(j=0;j<A;j++)
#define loopk(A) for(k=0;k<A;k++)
#define loopl(A) for(l=0;l<A;l++)
#define loopm(A) for(m=0;m<A;m++)
#define loopn(A) for(n=0;n<A;n++)
#define loopp(A) for(p=0;p<A;p++)
#define loopij(A,B) for(i=0;i<A;i++) for(j=0;j<B;j++)
#define MALLOC(A,B,C,D) if((A=(C *)malloc((B)*sizeof(C)))==NULL) { \
    fprintf(stderr, strcat(D,": insufficient memory\n")); \
    exit(-1); }
#define CREATE_FILE(A,B,C) if((A=fopen(B,"w")) == NULL) { \
       printf(strcat(C,": can't open for writing - %s.\n"),B); \
    exit (-1); }
#define OPEN_FILE(A,B,C) if((A=fopen(B,"r")) == NULL) { \
       printf(strcat(C,": can't open for reading - %s.\n"),B); \
    exit (-1); }
#define idx(I,J,N) (I)*(N)+(J)
#define FEQ(A,B) ( ( ((A)-(B)) < 1E-6 ) ? 1 : 0 )
#define IABS(A) ((int)((-(A)<(A))?((A)):(-(A))))

/** Dividing by 100 insures that cc and gcc give same results **/
#define IRAN1(A) ((int)(ran1(A)*(float)INT_MAX)/100)

/** All of these are dependent on the definition of "layer" **/
#define MAX_INPUTS          1500
#define MAX_NODES           50
```

```
#define MAX_H1_NODES          50
#define MAX_H2_NODES          50
#define MAX_OUTPUTS           1500

#define MAX_VECTORS           10000
#define STATE_SIZE          256

#define WTS_TYPE_MSF 2 /* new weights file */
#define WTS_TYPE_1   1 /* new weights file */
#define WTS_TYPE_0   0 /* old weights file */

#define TRAIN    0
#define TEST     1

#define THREE_LAYER   3
#define TWO_LAYER     2

#define MATRIX_ALLOCATOR(DATA_TYPE,FCN_NAME) \
DATA_TYPE **FCN_NAME(nrl,nrh,ncl,nch) \
int nrl,nrh,ncl,nch; \
{ \
  int i; \
  DATA_TYPE **m; \
          \
  m=(DATA_TYPE **) malloc((unsigned) (nrh-nrl+1)*sizeof(DATA_TYPE*)); \
  if (!m) nrerror("allocation failure 1 in matrix()"); \
  m -= nrl; \
        \
  for(i=nrl;i≤nrh;i++) { \
    m[i]=(DATA_TYPE *) malloc((unsigned) (nch-ncl+1)*sizeof(DATA_TYPE)); \
    if (!m[i]) nrerror("allocation failure 2 in matrix()");\
    m[i] -= ncl; \
  } \
  return m; \
}

#define MATRIX_FREE(DATA_TYPE,FCN_NAME) \
void FCN_NAME(m,nrl,nrh,ncl,nch)      \
DATA_TYPE **m;                 \
int nrl,nrh,ncl,nch;          \
{                         \
  int i;                \
                        \
  for(i=nrh;i≥nrl;i--) free((char*) (m[i]+ncl)); \
  free((char*) (m+nrl));        \
}
```

A.11 globals.h

```
/**** Global Variables ***/
 extern int num_inputs, hide_one, hide_two, num_outputs, temp_outputs;
 extern double eta_in, eta_out, eta_H1_H2;
 extern int net_type;
 extern int wts_type;
 extern int num_layers;
 extern int iterations ;
 extern int initial_seed, part_seed, trn_seed;
 extern int num_states;
 extern int num_vectors ;
 extern int *ftr_list;
 extern int *vec_num;
 extern int *vec_entry;
 extern int num_trn , *trn_list;
 extern int num_tst , *tst_list;
 extern int norm_type;
 extern int dominant_sensor ;
 extern int num_flir, *flir_list;
 extern int num_rng, *rng_list;
 extern int current_vector;
 extern int max_iterations, output_interval;
 extern int batch_size , batch_cnt ;

 extern char state[STATE_SIZE];

 extern double trn_frac;
 extern double alpha;
 extern double max_value;

extern struct layer {
    int       num_inputs;
    int       num_outputs;
    double    eta;
    double    **w;
    double    **dw;
    double    **w_old;
    double    *theta;
    double    *dtheta;
    double    *theta_old;
    double    *del;
    double    *beta;
    double    *gamma;
    double    *mask;
    double    *X;
    double    *Y;
        } L1, L2, L3;
 extern double *input_mask;
 extern double *xhat;
 extern double D_out[MAX_OUTPUTS];
 extern struct layer *output_layer, *Layer[3];
 extern double **db_in;
 extern double **db_out;
 extern double **id_out;
 extern double *mean, *sd;

/** Global Data for Kalman Training **/
extern double    **R, d[MAX_OUTPUTS], z[MAX_OUTPUTS];
```

A.12 makedata.c

```
/******************************************************************************
   NAME: makedata.c
   INVOKED: makedata classlist list_size Imagesize #_classes outputfile
   DATE: 25 May 92
   DESCRIPTION: This routine generates the data file used by the network.
   WRITTEN BY: Dennis L. Krepp
   MODIFIED:
   SUBROUTINES CALLED:
   FUTURE MODIFICATIONS/BUGS:
******************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#define PIXELS   1024
/****************************************
To use larger images change the size of
PIXELS above.
****************************************/
int image[PIXELS];

main(argc,argv)
int argc;
char *argv[];

{
  FILE *facein, *fout, *classfile;
  int data, M, k, j;
  int class, i, inputs, outputs;
```

```
   int temp;
   char outfile[30], filename[30], tempfile[30];
   char *strcpy();

if (argc ≠ 6) {
  printf("!!! The command line should be !!!:\n\n makedata classlist_file  #_files_in_list  Image_size(pixels)
#_classes  Output_file_name\n");
   exit(0);
}
/****************** Set Up Files ***********************/

if ((classfile = fopen(argv[1], "r")) == NULL)
 {
   printf("I can't open the classlist_file");
   fflush(stdout);
   exit(−1);
 }

M=atoi(argv[2]); /* M = Number of images in classlist_file */
inputs = atoi(argv[3]);
outputs = atoi(argv[4]);
strcpy(outfile, argv[5]);


/****** Open output file for writing *****/

if ((fout = fopen(outfile,"w")) == NULL)
 {
   printf("I can't open the output file %s \n",outfile);
   fflush(stdout);
   exit(−1);
 }
printf("Output file:  %s \n\n",outfile);fflush(stdout);

fprintf(fout, "%d\n",inputs);
fprintf(fout, "%d\n", outputs);

/*************************************************
1.  Read data_list_file for filename
2.  Write exemplar number to output file
3.  Copy input file to output file
*************************************************/

for(k=1; k≤M; k++)
 {

   fscanf(classfile, "%s\n", filename);
   printf("Input file:  %s \n", filename);fflush(stdout);

   fscanf(classfile, "%d", &class);
   printf("Class is:   %d\n", class);

   fprintf(fout, "%d\n", k);
   printf("Exemplar number:  %d\n\n", k);
   fflush(stdout);


   if ((facein = fopen(filename, "r")) == NULL)
   {
   printf("I can't open the input file");fflush(stdout);
   exit(−1);
   }

   while (fscanf(facein, "%d", &temp) == 1)
   {
   fprintf(fout, "% 8.4f ",(float)temp);
   }

/**** Write Class data to file ****/

   for(j=outputs;j≥1;j−−)
    {
    if(class == j)
```

```
      fprintf(fout,"% f ", 0.90000);
    if(class ≠ j)
      fprintf(fout,"% f ", 0.10000);
    }
   fprintf(fout, "%s", "\n\n");

   fclose(facein);
 }

fclose(fout);
fclose(classfile);
printf("\n!!!!! ALL DONE !!!!!\n\n");

}
```

Bibliography

1. Abbas, H. and M. Fahmy. "A Neural Model for Adaptive Karhunen-Loéve Transformation (KLT)," *IJCNN, II*:975–980 (1992).
2. Aibara, Tsunehiro *et al.* "Human face recognition by P-type Fourier descriptor," *SPIE Visual Communications and Image Processing, 1606*:198–203 (1991).
3. Baldi, Pierre and Kurt Hornik. "Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima," *Neural Networks, 2*:53–58 (1989).
4. Bouattour, *et al.* "Neural Nets for Human Face Recognition," *IEEE IJCNN, III*:700–704 (June 1992).
5. Cottrell, Garrison W. and Janet Metcalfe. *EMPATH: Face, Emotion and Gender Recognition Using Holons*. 2929 Campus Drive, San Mateo, CA, 94403: Morgan Kaufmann Publishers, Inc., 1991.
6. Cottrell, Garrison W. and Paul Munro. "Principal component analysis of images via back propagation," *SPIE Visual Communications and Image Processing, 1001*:1070–1077 (1988).
7. Cottrell, Garrison W. *et al.* "Learning Internal Representations from Gray-Scale Images: An Example of Extensional Programming," *Proceedings of the Ninth Annual Cognitive Science Society Conference, Volume unknown*:461–473 (1987).
8. Damasio, Antonio R. "Prosopagnosia," *Trends in Neuroscience, 8*:132–135 (1985).
9. Duda, Richard O. and Peter E. Hart. *Pattern Classification and Scene Analysis*. New York: John Wiley and Sons, 1973.
10. Farahati, Nader *et al.* "Real-time recognition using novel infrared illumination," *Optical Engineering, 31(8)*:1658–1662 (August 1992).
11. Fleming, Michael K. and Garrison W. Cottrell. "Categorization of Faces Using Unsupervised Feature Extraction," *IEEE International Joint Conference on Neural Networks, 2*:65–70 (1990).
12. Foldiák, Peter. "Adaptive Network for Optimal Linear Feature Extraction," *IEEE International Joint Conference on Neural Networks, 1*:401–405 (1989).
13. Gay, Kevin P. *Autonomous Face Recognition.* MS thesis, AFIT/GE/ENG/92D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.
14. Geschwind, Norman. "Specializations of the Human Brain," *Scientific American*, 107–120 (September 1979).
15. Goble, James R. *Face Recognition Using the Discrete Cosine Transform.* MS thesis, AFIT/GE/ENG/91D-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
16. Harmon, *et al.* "Machine Identification of Faces," *Pattern Recognition, 13*:97–110 (1981).
17. Kung, S. and K. Diamantaras. "A Neural Network Learning Algorithm for Adaptive Principal Component Extraction (APEX)," *IEEE ICASSP, 1*:861–864 (1990).
18. Lambert, Lawrence C. *Evaluation and Enhancement of the AFIT Autonomous Face Recognition Machine.* MS thesis, AFIT/GE/ENG/87D-35. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

19. Lei, Xu. and Alan Yuille. "Robust PCA Learning Rules Based on Statistical Physics Approach," *IEEE IJCNN, I*:812–817 (June 1992).

20. Meadows, J. C. "Varieties of Prosopagnosia," *Journal of Neurology, Neurosurgery, and Psychiatry*, 498–501 (1974).

21. Nakagawa, S.*ēt al.* "Dimensionality Reduction of Dynamical Patterns using a Neural Network," *Advances in NIPS, unk*:unk (1990).

22. Oja, Erkki. "A Simplified Neuron Model as a Principal Component Extractor," *Journal of Mathematical Biology, 15*:267–273 (1982).

23. Oja, Erkki. "Data Compression, Feature Extraction, and Autoassociation in Feedforward Neural Networks," *Artificial Neural Networks, unk*:737–745 (1991).

24. Oja, *et al.* "Learning in Nonlinear Constrained Hebbian Networks," *Artificial Neural Networks, unk*:385–389 (1991).

25. Payne, Tanya *et al.* "Backpropagation Neural Networks for Facial Verification Update," *Los Alamos National Laboratory* (1992 Unpublished).

26. Press, *et al. Numerical Recipes In C.* Cambridge: Cambridge University Press, 1988.

27. Robb, Barbara C. *Autonomous Face Recognition Machine Using a Fourier Feature Set.* MS thesis, AFIT/GE/ENG/87D-35. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

28. Rolls, *et al.* "The effect of learning on the face selective responses of neurons in the cortex in the superior temporal sulcus of the monkey," *Experimental Brain Research, 76*:153–164 (1989).

29. Ruck, Dennis W. *Characterization of Multilayer Perceptrons and their Application to Multisensor Automatic Target Detection.* PhD dissertation, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

30. Runyon, Kenneth R. *Face Recognition System.* MS thesis, AFIT/GE/ENG/92D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1992.

31. Russell, Robert L. *Performance of a Working Face Recognition Machine Using Cortical Thought Theory.* MS thesis, AFIT/GE/ENG/85D. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1985.

32. Samal, Ashok and Prasana A. Iyengar. "Automatic Recognition and Analysis of Human Faces and Facial Expressions: A Survey," *Pattern Recognition, 25*:65–77 (1992).

33. Sander, David D. *Enhanced Autonomous Face Recognition Machine.* MS thesis, AFIT/GE/ENG/89D-19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989.

34. Sanger, Terence. "Optimal Unsupervised Learning in a Single-Layer Linear Feedforward Neural Network," *Neural Networks, 2*:459–473 (1989).

35. Smith, Edward J. *Development of an Autonomous Face Recognition Machine.* MS thesis, AFIT/GE/ENG/86D-36. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

36. Suarez, Pedro F. *Face Recognition with the Karhunen-Loeve Transform.* MS thesis, AFIT/GE/ENG/91D-54. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

37. Tarr, Gregory L. *Multi-Layered Feedforward Neural Networks for Image Segmentation.* PhD dissertation, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

38. Tou, Julius C. and Rafael C. Gonzalez. *Pattern Recognition Principles.* Reading, MA: Addison-Wesley Publishing, 1974.

39. Turk, Matthew A. and Alex P. Pentland. "Eigenfaces for Recognition," *Journal of Cognitive Neuroscience,* 1–28 (September 1990).

40. Turk, Matthew A. and Alex P. Pentland. "Recognition in Face Space," *SPIE Intelligent Robots and Computer Vision IX: Algotithms and Techniques, 1381*:43–54 (1990).

41. Valentine, Tim and Andre Ferrara. "Typicality in Categorization, recognition and identification: Evidence from face recognition," *British Journal of Psychology, 82*:87–102 (1982).

42. Wu, Chyuan Jy and Jun S. Huang. "Human Face Profile Recognition by Computer," *Pattern Recognition, 23*:255–259 (1990).

Vita

Captain Dennis L. Krepp was born on March 6, 1958 in Ephrata, Pennsylvania. He graduated from Warwick High School in Lititz, Pennsylvania in 1976. Capt. Krepp entered the Air Force in May, 1978 as a Munitions Systems Specialist. He served three years at Davis-Monthan AFB, Arizona with the 355th Equipment Maintenance Squadron, and three years at Lowry AFB, Colorado with the 3460 Technical Training Wing. He entered the Airman's Education and Commissioning Program in August, 1984 and completed a Bachelor of Science degree in Electrical Engineering at the University of Colorado in May, 1987. He served three years with the Electronic Systems Division at Hanscom AFB, Massachusetts before entering the School of Engineering, Air Force Institute of Technology in June, 1991. He is married to Karen (Muije) Krepp of Green River, Wyoming and has two children: Michelle Lynn, age 11, and R. Adam, age 9.

Permanent address: 1069 Furnace Hill Pike
Lititz, Pennsylvania 17543

REPORT DOCUMENTATION PAGE	Form Approved OMB No 0704 0188

[illegible]

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE December 1992	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE

Face Recognition With Neural Networks

5. FUNDING NUMBERS

6. AUTHOR(S)

Dennis L. Krepp, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/92D-23

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Maj Rodney Winter
Govt Agcy
9800 Savage Rd
Ft Meade, MD 20755-6000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This study investigated neural networks for face verification and classification. The research concentrated on developing a neural network based feature extractor and/or classifier to perform authorized user verification in a realistic work environment. Recognition accuracy, system assumptions, training time, and execution time were analyzed to determine the feasibility of a neural network approach. Data was collected using a camcorder and two segmentation schemes: manual segmentation and motion-based, automatic segmentation. Data consisted of over 2000, 32x32 pixel, 8 bit gray scale images of 52 subjects; each subject had two to ten days worth of images collected. Several training and test sets were created and then used to train and test the following networks: a backpropagation network using the raw data as inputs; a backpropagation network using Karhunen-Loève Transform coefficients, computed from the raw data, as inputs; and a backpropagation network using features extracted by an identity network as inputs. The classification networks performed well on constrained, single day captured, data bases but performed poorly on data gathered over multiple days . For multiple days, a verification network using a single hidden layer with backpropagation obtained 95% verification accuracy and is suitable for use in a face verification system.

14. SUBJECT TERMS face recognition, neural networks, identity networks, backpropagation, user verification	15. NUMBER OF PAGES 122
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL